

General Disclaimer

One or more of the Following Statements may affect this Document

- This document has been reproduced from the best copy furnished by the organizational source. It is being released in the interest of making available as much information as possible.
- This document may contain data, which exceeds the sheet parameters. It was furnished in this condition by the organizational source and is the best copy available.
- This document may contain tone-on-tone or color graphs, charts and/or pictures, which have been reproduced in black and white.
- This document is paginated as submitted by the original source.
- Portions of this document are not fully legible due to the historical nature of some of the material. However, it is the best reproduction available from the original submission.

JPL-SP-43-29

STANDARDIZED
DEVELOPMENT
OF
COMPUTER
SOFTWARE

N79-15676

Unclas
43357

G3/61

(NASA-CR-158070) STANDARDIZED DEVELOPMENT
OF COMPUTER SOFTWARE. PART 2: STANDARDS
(Jet Propulsion Lab.) 559 p HC A24/MF A01
CSCI 09E

Robert C. Tausworthe

PART II
STANDARDS

STANDARDIZED
:
DEVELOPMENT
:
OF
:
COMPUTER
:
SOFTWARE

Robert C. Tausworthe

PART II
STANDARDS

This material was prepared by the Jet Propulsion Laboratory under contract No.
NAS 7-100, National Aeronautics and Space Administration.

JPL SP 43-29, Part II, August 1978

PREFACE

At the time Part I of this work was being published, the Jet Propulsion Laboratory's Deep Space Network (DSN) was in the process of developing and writing a set of Software Standard Practices, for which Part I was cited as the "methodology textbook." The standards were developed over several years by a group we simply called the "Software Seminar." It was created and chaired in its "pathfinder period" by Walter K. Victor, who was, by the way, the significant inspirator of Part I. Mahlor F. Easterling then led that symposium (second Webster [43] meaning) through its next "pilot" phase. Edward C. Posner steered it through the arduous, major, final phase involving detailed standards development, consensus building, writing, review, and publication; he also sponsored the final symposium (first Webster [43] meaning) immediately after the standards were signed off by upper management. Daniel C. Preska administered the writing of the standards, with editorial assistance by Richard C. Chandlee.

The test-bed for the methodology reported in Part I had been an effort of medium magnitude—a program (the MBASICTM language processor) containing about 25,000 lines of non-real-time assembly language code. The results of that methodology test-bed seemed to indicate that programmer performance better than had been encountered in past DSN projects could be extended to the DSN as a whole—an organization involving perhaps 100 programmers in various disciplines.

With that belief, the implementation team manager of a critical hardware/software project—to completely upgrade the digital data systems in all of the deep-space stations around the world—undertook the additional task of applying and evaluating the then-emerging DSN Software Standard Practices as a standards-test-bed activity. The overall project, including software for system performance tests, generated approximately 100,000 lines of hard-real-time assembly language code over about 2-1/2 years.

That project could ill-afford to be a mere guinea pig for a software standards seminar, because the delivery of the first-phase system was crucially tied to upcoming spacecraft launch dates and committed on-going

iv Preface

missions. Even moderate deviations from the original schedule could not be tolerated. Short slippages could, perhaps, be accommodated if detected early enough for appropriate replanning.

Yet the prospects for success using the standards seemed good. A software manager was appointed, cognizant software development engineers for each of the major assemblies making up the system were selected, and a secretariat function (Chapter 17) was established. Subcontractors were selected to aid in all activities of the implementation and integrated with JPL personnel into a unified team. All were admonished to apply and conform to the (draft-form) software standards to the maximum extent, except where it could be shown that adherence to standards was interfering with the schedule. Waivers were granted on a case-by-case basis, in writing, to record the details wherein standards proved ineffective.

The project demonstrated numerous gratifying benefits arising from the methodology presented in Part I and the more detailed standards contained in this second volume. Among these were good schedule and cost performance, high product reliability, adequate documentation, increased productivity, and smooth development and delivery.

The delivery date did slip from the original 2-1/2 year plan by somewhat less than 1 month (3% accuracy of original plan). However, this slip was predicted about 7 months in advance of its actual occurrence, so that effective contingency planning could be initiated. The 6% cost overrun was also predicted well enough in advance that reallocation of project resources was effective. These excesses were considered unusually slight, particularly in comparison to past JPL experience and then-current industry-published data.

The software contained an average of approximately 3 errors per thousand lines of code, measured from the beginning of system integration tests, as compared to 10-20 errors per thousand lines commonly reported in similar projects not employing top-down structured programming methodology. This test phase, as a matter of fact, required only about 15% of the overall effort, whereas industry-published figures and previous JPL experience quoted about a 50% level of effort. The difference in effort was expended in the design and planning phases to produce a more mature, well-documented, reliable product.

The development and delivery were reported as being smooth and controlled. No "tiger teams" were required during implementation, no significant renegotiation of software commitments was needed near the end

of production, and the software was delivered ready for operation with very few liens levied for future corrective action.

The standards, of course, did not accomplish these achievements—*people* did. JPL was fortunate to have had outstanding personnel performing in an exceptional, professional manner throughout the project. All that one may claim for the standards is that they provided a methodology which allowed each member of the project to apply himself or herself toward the accomplishment of project goals in the most effective way.

That methodology held up to its promise. The managers, designers, coders, operational personnel, documentarians, and theoreticians in concert had crafted and codified a viable, detailed set of practices for producing software. All concerned had had a voice in the creation and adjustment of their software engineering discipline, and for once the "horse" designed and built by a "committee" didn't turn out to be a "camel."

Part II of this monograph, then, exposes this detailed set of rules for software implementation. I have broadened some of the DSN practices in some instances, in an attempt to make them more readily adaptable to organizational structures different than that of the DSN. Additional consonant practices from other sources have also been incorporated to broaden the scope of applicability to projects of types other than the high-technology, high-efficiency, single-purpose, custom-built variety demanded by the deep-space-station environment.

There are many whom I must thank and acknowledge for their many and various contributions toward the completion of this second volume. Robertson Stevens, the former manager of a large computing facility and, during this time, manager of the upper-level organization containing the hardware/software project, was the propounder of many of the management policies and status monitors that are found in this work. Paul T. Westmoreland was the manager of the implementation project; his professionalism, ability to manage, faith in a standardized approach, and courage to commit that approach to a critical task have been a personal inspiration.

I must also acknowledge the effectiveness of Alvin F. Ellman of the Bendix Corporation, who was software manager. It was perhaps Al's ability to recognize what quantitative information a programmer could communicate naturally to management and others that led to the refined status monitors that proved so effective. His ability to relate to and interface with project-internal programmers and project-external systems engineers and users was a major factor in an organization-wide feeling of confidence in the health of the growing and maturing software.

vi Preface

The subsystem Cognizant Development Engineers were Robert Desens, Frank Hlavaty, Ronald Murray, Gary Osborn, and Steve Yee. Observance of their applications of the standards and their performance under the standards produced many refinements for effectiveness.

The members of the DSN Programming System Steering Committee included, at various times, Walter K. Victor, Robertson Stevens, Mahlon Easterling, Lee W. Randolph, Carl W. Johnson, Cecil P. Wiggins, Edward C. Posner, Malvin L. Yeater, William C. Frey, William D. Hodgson, Angela Irvine, Raul D. Rey, Richard B. Miller, and Donald L. Gordon. Each made special contributions too numerous to single out.

R. Booth Hartley and Lawrence R. Hawley were both co-developers and appliers of the rules given here during the various implementations of elements of the DSN Programming System. Their support, feedback, and ability were sorely needed and freely provided during the preparation of this material. Kay Landon and Leonard Benson proofread Part I and generated its index; they also programmed a prototype CRISPFLOW processor, leading to the descriptions in Appendix G. Annamarie Grana helped evaluate the utility of Appendix C, using it as a guide for the generation of two SRDs. Frank Hlavaty collaborated in the formation of Appendix E. Michelle Martin and Marshall Polsley contributed to the format and content of Appendix I. Richard Schwartz's influence is prevalent in the standard language discussions in Chapter 17. John Johnson and Henry Kleine were instrumental in the formation of the CRISP language.

I give special thanks to Georgiana Clark, who typed and corrected the entire manuscript; to Carol Rosner, who had typed a preliminary draft of the first five chapters; to Margaret Seymour, who drafted all the figures except those in Appendices G and L; to Silvia Munoz, who aided in generating the index; to Harold Yamamoto, who edited the volume for publication; and to Doris Perry, who coordinated all the artwork and was responsible for the final page makeup. I also extend a belated thanks to Anita Sohus, who coordinated all the artwork for Part I.

Finally, I wish to thank those who have participated in the many seminars and classes given on this material during its various stages of completion; many insights into the secrets of software engineering across a broad programmer base were revealed to me as the result of these interactions.

Robert C. Tausworthe

CONTENTS

PART II

XI. SOFTWARE REQUIREMENTS AND DEFINITION STANDARDS	1
11.1 GENERATING SOFTWARE REQUIREMENTS	2
11.2 GENERATION OF THE SOFTWARE ARCHITECTURAL DESIGN	7
11.3 GENERATING THE SOFTWARE FUNCTIONAL SPECIFICATION	14
11.4 DOCUMENTING TECHNICAL REQUIREMENTS AND FUNCTIONAL SPECIFICATIONS	19
11.5 RULES FOR THE SOFTWARE DEVELOPMENT LIBRARY	33
11.6 SUMMARY	34
XII. PROGRAM DESIGN AND SPECIFICATION STANDARDS	35
12.1 RULES FOR STRUCTURAL DESIGN	36
12.2 RULES FOR DATA STRUCTURING AND RESOURCE ACCESS DESIGN	39
12.3 RULES FOR DEVELOPING STRUCTURED PROGRAMS	45
12.4 RULES FOR APPLYING STRUCTURED PROGRAMMING THEORY	49
12.5 RULES FOR REAL-TIME STRUCTURED PROGRAMS	53
12.6 STANDARD DESIGN PRACTICES	57
12.7 RULES FOR DOCUMENTING STRUCTURED SPECIFICATIONS	58
12.8 RULES FOR THE SOFTWARE DEVELOPMENT LIBRARY	83
12.9 SUMMARY	84
XIII. PROGRAM CODING STANDARDS	85
13.1 RULES FOR CODING STRUCTURED PROGRAMS	86

viii Contents

13.2	RULES FOR CODING STRUCTURED REAL-TIME PROGRAMS	93
13.3	RULES FOR DOCUMENTING STRUCTURED CODE	94
13.4	STANDARD PRODUCTION PROCEDURES	98
13.5	SUMMARY	101
XIV.	DEVELOPMENT TESTING STANDARDS	103
14.1	RULES FOR SPECIFYING DEVELOP- MENT TESTS	104
14.2	RULES FOR DEVELOPING TESTS FOR REAL-TIME PROGRAMS	107
14.3	RULES FOR ASSEMBLING AND PER- FORMING TESTS	107
14.4	RULES FOR CODING TEST ELEMENTS	109
14.5	RULES FOR DOCUMENTING DEVELOPMENT-TEST SPECIFICATIONS.	111
14.6	RULES FOR DOCUMENTING TEST RESULTS	111
14.7	RULES FOR THE SOFTWARE DEVELOPMENT LIBRARY	112
14.8	DIAGNOSTIC PROCEDURES	113
14.9	SUMMARY	114
XV.	QUALITY ASSURANCE STANDARDS	115
15.1	STANDARD QA ACTIVITIES	116
15.2	QA MEASURES DURING PROGRAM DEVELOPMENT.	117
15.3	SOFTWARE TESTING CHARACTERISTICS	118
15.4	RULES FOR ACCEPTANCE TESTING AND CERTIFICATION	129
15.5	SOFTWARE AUDITS	132
15.6	DOCUMENTATION OF QA ACTIVITIES	138
15.7	RULES FOR SECURITY, INTEGRITY, AND CONFIGURATION CONTROL	143
15.8	SUMMARY	145
XVI.	LEVELS OF DOCUMENTATION	147
16.1	HUMAN FACTORS	148

16.2	DOCUMENTATION STANDARDS.	155
16.3	PREPARATION OF DOCUMENTATION . . .	166
16.4	SUMMARY	169
XVII.	A STANDARD SOFTWARE PRODUCTION SYSTEM	171
17.1	AN INTEGRATED SOFTWARE PRODUCTION SYSTEM	172
17.2	THE STANDARD PRODUCTION SYSTEM SUPPORT LIBRARY	185
17.3	STANDARD PROGRAMMING LANGUAGES AND LANGUAGE STANDARDS	187
17.4	CRISP-PDL PROCESSING	201
17.5	FLOWCHARTING FROM CRISP-PDL . . .	205
17.6	TEXT AND PROGRAM FILE EDITING. . .	210
17.7	MANAGEMENT DATA AND STATUS REPORTING	212
17.8	CONCLUSION	217
APPENDICES		
A.	GLOSSARY OF TERMS AND ABBREVIATIONS	219
B.	STANDARD FLOWCHART SYMBOLS . . .	237
C.	SOFTWARE REQUIREMENTS DOCUMENT TOPICS	251
D.	SOFTWARE DEFINITION DOCUMENT OUTLINE	263
E.	SOFTWARE SPECIFICATION DOCUMENT OUTLINE	275
F.	USER INSTRUCTION MANUAL TOPICS . .	295
G.	CRISP SYNTAX AND STRUCTURES . . .	309
H.	DEVELOPMENT PROJECT NOTEBOOK CONTENTS	373
I.	OPERATIONS MANUAL CONTENTS . . .	383
J.	SOFTWARE TEST REPORT CONTENTS . .	399
K.	SOFTWARE MAINTENANCE MANUAL CONTENTS	407
L.	SAMPLE PROGRAMS FOR PROJECT MANAGEMENT	415
M.	USEFUL STANDARD FORMS	513
REFERENCES		539
INDEX		543

PART I

- I. Introduction
- II. Fundamental Principles and Concepts
- III. Specification of Program Behavior
- IV. Program Design
- V. Structured Non-Real-Time Programs
- VI. Real-Time and Multiprogrammed Structured Programs
- VII. Control-Restrictive Instructions for Structured Programming (CRISP)
- VIII. Decision Tables as Programming Aids
- IX. Assessment of Program Correctness
- X. Project Organization and Management

Part II

STANDARDS

XI. SOFTWARE REQUIREMENTS AND DEFINITION STANDARDS

This chapter is the first of a set containing specific standards extracted from, or generated in response to, the methods presented in Part I. These are the rules that guide the top-down, hierarchic, modular, structured approach to software development.

There are, of course, no universal rules to make intricate programming a simple task, and there is perhaps very little hope of ever completely formalizing the programming process. Design is a creative, inventive craft. But merely identifying the constraints, objectives, design tools, and parameters in a standardized way yields considerable progress in dealing with problems effectively. Furthermore, these standard procedures can be taught. References [1] through [6] are examples of standards in effect based on the methodology reported here.

The standards and practices contained in the remainder of this work are meant primarily to apply to new programs or major extensions to existing programs intended for operational use. They are meant to be easy to use, to

be somewhat flexible, and to provide guidelines for focusing the activities toward what is most needed.

The use of a consistent outline and format for documenting each activity is presumed. The outlines in Appendices C, D, and E contain a detailed set of topics to be considered in defining the requirements and functional behavior of a software package. The topics also give guidelines as to what material is to be specified within each topic.

A large portion of any software engineering activity deals fundamentally with the planning of a software development, rather than the actual doing of it. I recognize that a discipline for such planning is needed just as much as a discipline for doing, so I have oriented the rules given here toward the more technical aspects of project engineering and software development. The interested reader wishing to delve more deeply into management and planning disciplines may consult [1] through [11].

11.1 GENERATING SOFTWARE REQUIREMENTS

A software requirement is a need established for a piece of software by an organization in order to achieve certain goals. The requirement-generation activity culminates in the approvals, negotiations, and commitments of resources necessary to initiate, sustain, and complete the software development. Although I have not considered requirements generation in past chapters to be among the software development activities, nevertheless, there are a few guidelines that can make the generation of software requirements more uniformly responsive to the needs of oncoming activities.

The Software Requirements Document (SRD) is, relatively speaking, a non-technical document; in its first-reviewed form, it probably contains only enough functional and technical information and requirements (perhaps by reference) to identify the need for a perhaps intangible capability. At this point, its content is primarily oriented so as to allow the authorizing organization to determine what it is approving.

Part of this approval involves the expenditure of resources to permit the requirers (and, later, implementors) to supply more planning information and technical detail.

Requirements are only definite to the extent that they are documented. The needed output of the requirement-generation activity is an SRD satisfying the following criteria (see Section 3.3):

- a. It should be adequate to identify the objectives of the program, its environment, the configuration needed for its operation, the resources required for its support, and the advantages and disadvantages in the service it provides, as related to the customer organization.
- b. It should be adequate to permit the developmental activities to proceed under a reasonable assurance that major revision of requirements will not be necessary.
- c. It should be adequate for review and approval by cognizant authority on the basis of its conceptual feasibility in accordance with the other criteria above. It should contain manpower, schedules, and development-cost estimates, as well as reasonable variances for these estimates, at least for the next phase of activity.

As I indicated in previous chapters, it may not always be feasible to actually complete the SRD until after some of the software development process has already begun (in a top-down way, of course). That part upon which the funding and manpower approvals are based (the overview) probably derives in largest part from the justification section (see Appendix C). This justification—intended for management—contains material oriented principally toward establishing the need for, and feasibility of, software to fulfill certain goals or missions of the funding organization.

The remainder of the SRD—for guiding the implementation—deals with setting forth technical requirements, developmental constraints, and acceptance criteria in enough detail to identify the external functional and operational characteristics of the software. These can subsequently be negotiated, refined, and then implemented so as to satisfy the sponsor's goals. The final SRD constitutes an agreement between the requesting and implementing organizations on the software to be produced.

The SRD contains material that may well be broken into several separate documents, such as, perhaps, a Software Justification Report, a Software Acquisition Plan, a Software Functional Requirement, and so on. Some material may be included directly, if not extensively, while some may be appended or referenced. The hierarchic nature of the outline in Appendix C permits this to be done quite easily in the most accommodating way, should the need arise.

The reader may notice, comparing Appendices C, D, and E, that the SRD, Software Design Definition (SDD), and Software Specification Document (SSD) outlines are all very similar in appearance. But while they cover the same general topic, they do not generate the same content. The SRD, upon completion, contains customer/user requirements and constraints, and estimates the resources available or needed for software

production. The SDD identifies those external characteristics of a program needed to fulfill the given requirements, establishes the program architecture, defines costs and schedules, and presents a work breakdown structure by tasks. The SSD defines the external and operational characteristics of the program and specifies how these are to be effected into internal program structures, "as built." By giving these documents the semblance of a conformable outline, I have tried to ensure a rough downward traceability from pre-design requirement to definitions, to implemented specifications, and upward, again, in the reverse order.

11.1.1 Rules for Generating Software Requirements

The following guidelines for structuring software requirements form a small set of standards to aid in the preparation of the SRD through the overview phase, culminating in a Requirements Review. Sections 3.1 and 3.2 contain useful guidelines for recognizing requirements.

1. Complete the SRD, supplying the material indicated in the topical outline (Appendix C), documented subject to the rules in Section 11.4.

Delete items in the outline that are irrelevant or do not apply, or mark them as "not applicable." Mark purposely unspecified items as "development prerogative," perhaps conditioned by the addition of modifiers, such as "subject to approval." Cite existing material, such as policies and constraints, by reference; include deviations from these, however. Figure 11-1 presents a graphic table of contents for the SRD that emphasizes the hierarchy of management information. The complete graphic table of contents is shown as Figure C-1 (Appendix C).

2. Establish a set of criteria and weights for the SRD review, such as breadth of coverage, level of detail, adherence to standards, etc., and obtain concurrence on these from the review board.

3. Identify cost and schedule drivers likely to impact the decisions that have to be made. Establish priorities and constraints for development, with emphasis placed on factors that tend to drive costs up if they are not identified and frozen early.

Include any known factors involving special complexity that tend to jeopardize schedules or place stringent demands on specific (and possibly scarce or unavailable) personnel.

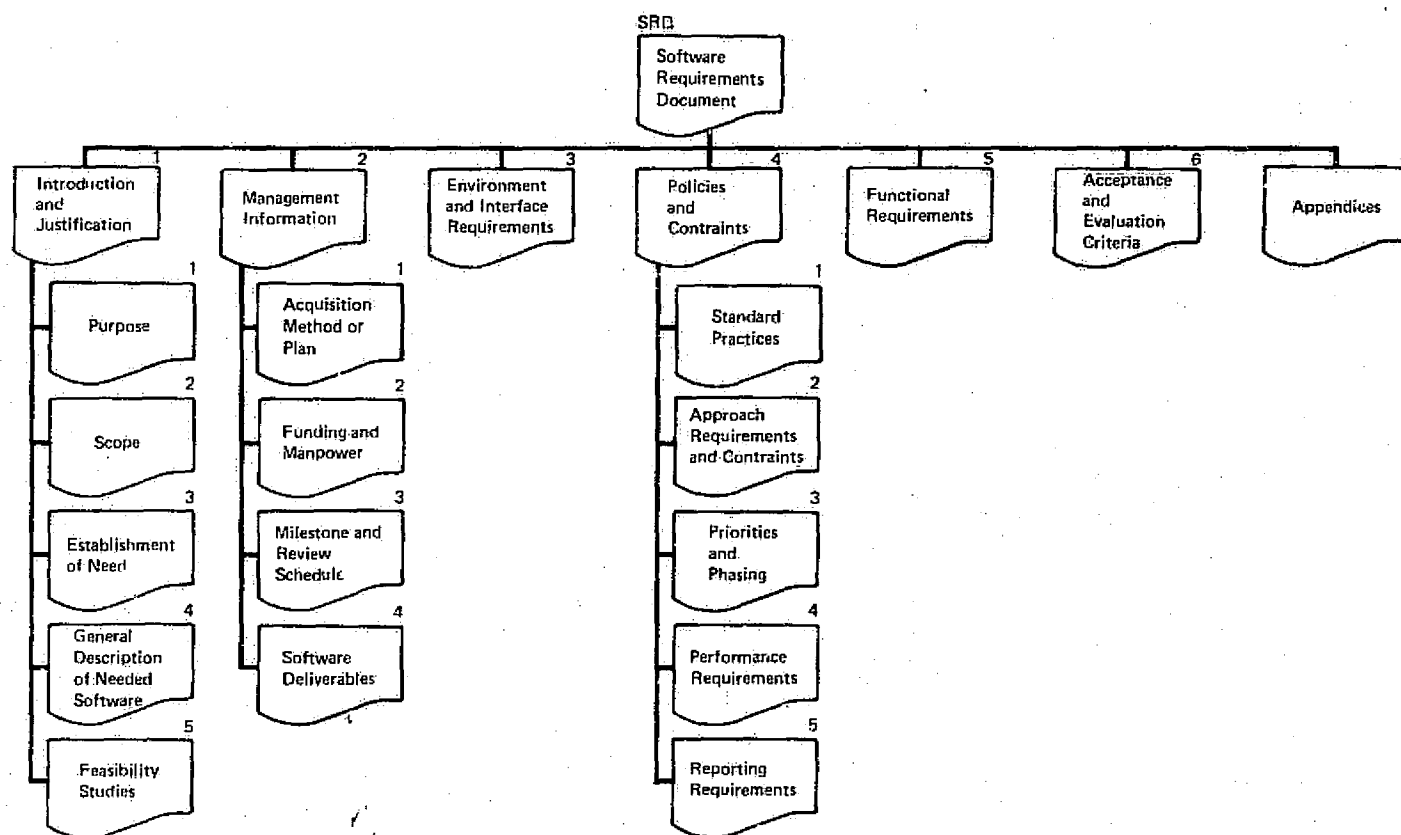


Figure 11-1. A visual table of contents for the SRD, showing hierarchy of management items important for approval phase

4. Identify and summarize or reference the functional processing of inputs required to obtain the proper outputs. Define user/customer technical requirements only down to that degree of detail beyond which no major influence on resources, schedule, or programmatic requirements is typically felt.

State specific input/output formats, details on internal or external data structures, computer internal operations, etc., only if they do exert a major influence on development. Generate and append requirements not in this category, but needed by implementors, as later detailed Functional Requirements.

5. Identify existing known capability, technology, or routines that may be used (directly or with modification), especially if the existence of these can have a great impact on cost or schedule.

6. Identify and rank characteristics that compete for development resources. First- and second-order dominances (Appendix L) are useful for this purpose. Qualify, as appropriate, any conditions that may alter this ranking.

7. Identify organizations or individuals (if known) to be involved in the implementation, operation, and use of the software, and state the roles of each.

8. Establish criteria for evaluating implementation team performance and for program acceptance and delivery.

9. Determine schedules and cost bounds. Estimate the relative values of required capabilities and available resources that can be applied toward delivery within these bounds.

For example, the earliest date that the software can be accommodated, if delivered; the latest operational date software can be delivered without serious degradation of the project goals; etc. Include possible conflicts between this activity and other ongoing work. This cost/time schedule is not intended to define a day-to-day work plan, but, instead, to establish a baseline for needed capability, to be refined in the next phase (the architectural design), based on more detailed considerations.

10. Prioritize technical requirements so as to accommodate the schedule and cost bounds.

For example, require program modes that will permit, perhaps, a degraded, but acceptable, partial performance during mission operations if delivery is late.

11. Prior to the actual SRD review, grade the SRD relative to review criteria and weights established in Rule 2, above, to gauge the pre-review quality of the SRD. After the review presentation, urge the review board likewise to grade the SRD.

12. Obtain management approval (at least informally or in principle) before proceeding to the next phase (the actual software development activity). Approval signatures should be those required by the normal organizational procurement policy, based on the estimated program cost, even if the work is to be performed in-house.

13. If iterations or negotiations of requirements are necessary after approvals were obtained in order to complete the SRD, inform the board-convening authority for possible re-review if there are major redirections or cost/schedule impacts.

14. Do not release any requirements for formal design and coding until these have been approved, with concurrence from the implementor.

11.2 GENERATION OF THE SOFTWARE ARCHITECTURAL DESIGN

Once a set of requirements has reached the implementing organization, the implementor enters upon a design feasibility study, translating the requirements given in the SRD into a definition of the needed design, the scope of work, refined cost/schedule estimates, and project organizational activities. This phase of the software definition activity culminates in a Software Design Definition* (SDD), which displays the program and development team architecture.

This activity can, and probably should, take place in concert with the generation of some of the later, more detailed, technical software requirements. Much of the SDD may even have to be developed as the SRD is being completed, as a cooperative interaction between initiator and implementor.

*This document is termed the Software Definition Document in the JPL Software Standards, cited in [1] through [6].

The SDD is, as was the SRD before it, largely non-technical, in that it contains a lot of planning information—this time from the implementor's point of view. It does, however, contain enough technical detail (perhaps by reference) to carry the program functional and internal behavioral specifications to the point of structuring the remainder of the development task according to the following criteria:

- a. It defines the manner in which the program and its technical documentation shall respond to each of the requirements in the SRD.
- b. It should be adequate to permit the remainder of the developmental activities to proceed under a reasonable assurance that a major revision of the program architecture will not be necessary.
- c. It should be adequate for review by technical authority and management on the basis of its technical feasibility and the soundness of its manpower, budgetary, and cost/schedule estimates.

A candidate outline for an SDD appears graphically in Figure 11-2 and more detailed in Appendix D. The rules for completing the SDD represent a bridge between those for the SRD and those for the detailed Software Functional Specification, described in Section 11.3, and those for the Programming Specification, the subject of Chapter 12.

11.2.1 The Work Breakdown Structure

The Work Breakdown Structure (WBS) is an enumeration of all work activities in hierarchic refinements of detail, which organizes work to be done into short, manageable tasks with quantifiable inputs, outputs, schedules, and assigned responsibilities. It may be used for project budgeting of time and resources down to the individual task level, and as a basis for progress reporting relative to meaningful management milestones. A software management plan based on a WBS contains the necessary tools to estimate costs and schedules accurately during the architectural phase, and to provide visibility and control during production.

Such a plan is structured to evaluate technical accomplishment on the basis of task and activity progress. Schedules and networks (see the first example of Appendix L) are built upon technical activities in terms of task milestones (i.e., accomplishments, outputs, and other quantifiable work elements). Projected versus actual task progress can be reviewed by technical audit and by progress reviews on a regular (say, monthly or bi-weekly) basis. Formal Project Design Reviews are major check points in this measurement system.

How refined should this WBS be? Let me answer this question by showing how the WBS and schedule projection accuracy are interrelated.

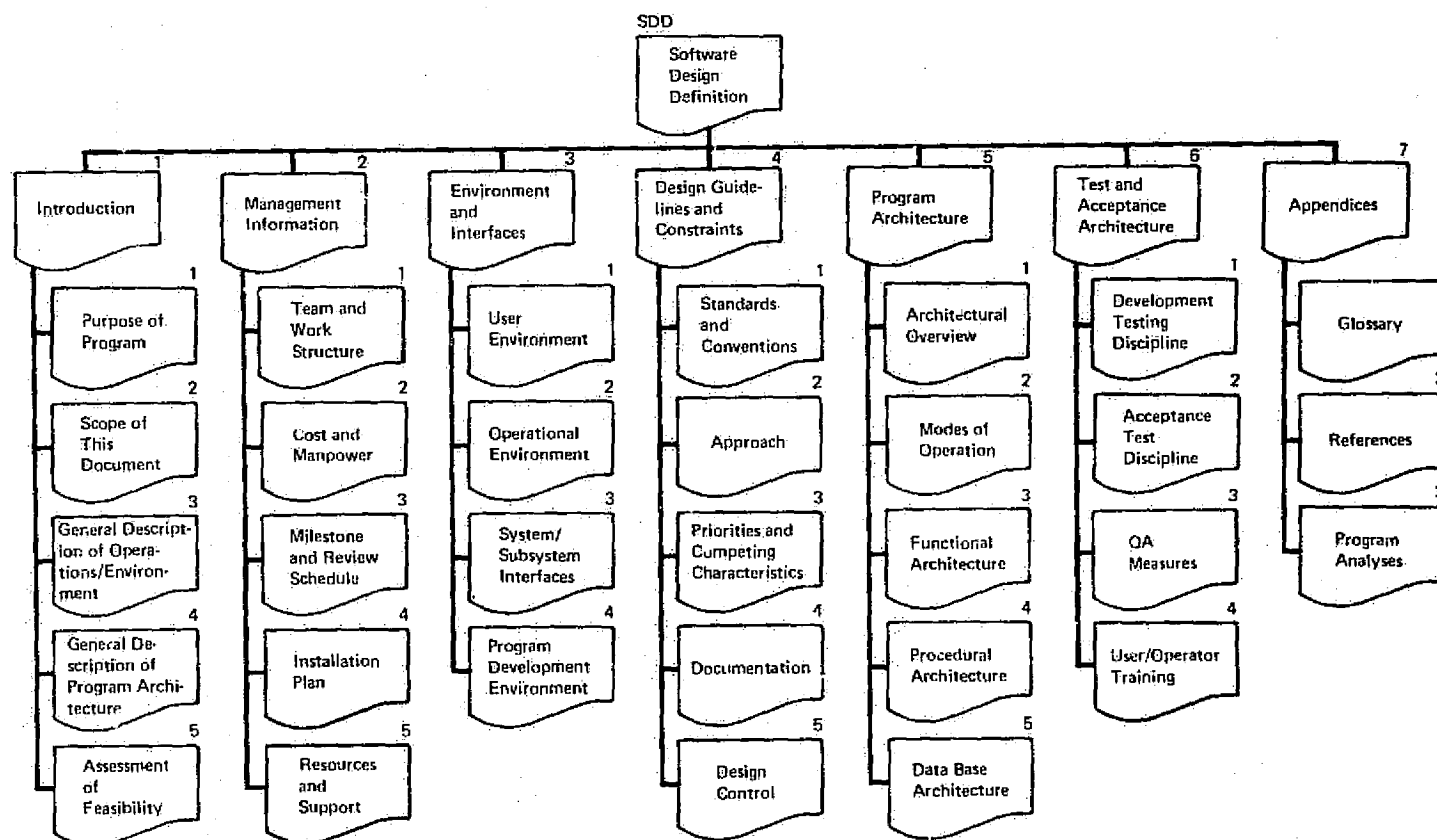


Figure 11-2. A hierarchic outline for the SDD for describing the program architectural design

If a project has identified, say, during its architectural design phase, a certain number of equi-effort milestones to be achieved during the course of development, then the mere number of milestones achieved by a certain date is an indicator of the progress toward that goal. A graph of accumulated milestones as a function of time, sometimes called a "rate chart," permits certain predictions to be made about the future completion date rather handily and with quantifiable accuracy, especially if the milestones are chosen properly.

Let us suppose that it is known *a priori*, as a result of generating the WBS, that a project will be completed after M milestones have been met. These milestones correspond to all of the tasks that have to be accomplished, and once accomplished are accomplished forever (i.e., some later activity does not re-open an already completed task; if such is the case, it can be accommodated by making M larger to include all such milestones as separate events). The number M , of course, may not be precisely known from the first, and any uncertainty in M is certainly going to affect the estimated completion date. This can be factored in as a secondary effect later, as needed for refinement of accuracy.

Now, let us further suppose that it has been possible to refine the overall task into these M milestones in such a way that each task is believed to require about the same amount of effort and duration to accomplish. Viewed at regular intervals (e.g., bi-weekly or monthly), a plot of the cumulative numbers of milestones reported as having been completed should rise linearly until project completion (Figure 11-3).

More quantitatively, let the reporting period interval be denoted T ; let n denote the most optimistic (but realistic, if all goes well) number of milestones (constant each ΔT period), which can be completed in ΔT days; let m be the average number actually completed; and let $q = 1 - (m/n)$. The last value, q , is the WBS deviation factor inherent in the WBS.

The value of m is a reflection of the team productivity and q is a measure of their ability to keep up with an optimistic version of the schedule. Both are attestations to team effectiveness—first, in their ability to produce, and, second, in their ability to create a work plan that adequately accounts for contingencies. It is important to stress that q does not in any way reflect on a team's ability to produce; it is merely a measure of how much the WBS depends on everything going well. Low values of q show low deviations from optimistic performance inherent in WBS formation, that's all.

The project with a given productivity should require a time $(M/m)\Delta T$ to complete, which time, of course, should not depend on whether a WBS was

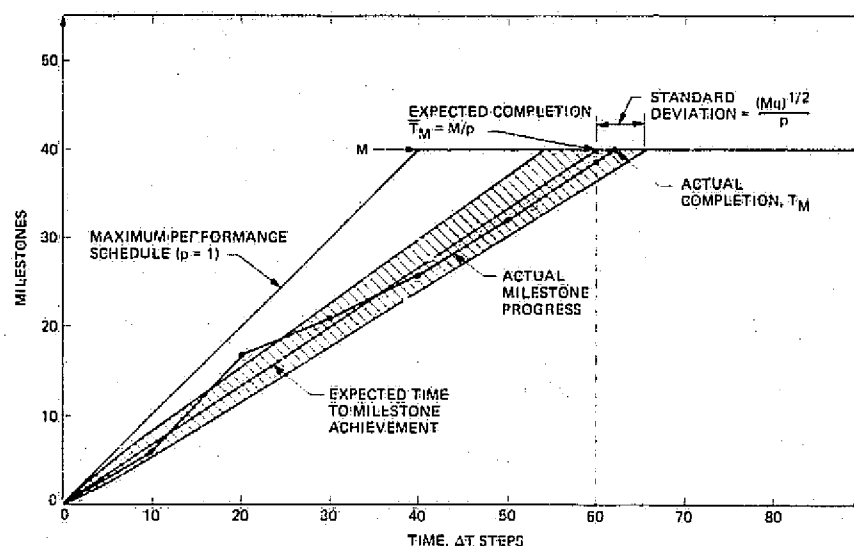


Figure 11-3. Cumulative milestones achieved as a function of time (case illustrated has parameters $p = 1 - q$, $n = 10$, $p = 2/3$, $M = 40$)

made or not (I am discounting, in this discussion, whether WBS generation increases or decreases productivity). Thus, M/m should be a constant value, relatively speaking. If M is made large, tasks are smaller and shorter, so proportionately more of them are completed each ΔT reporting period. The original project schedule will, in fact, assume some productivity, or mean accomplishment rate, but an actual performance value is generally unknown until progress can be monitored for some period of time.

But while the numbers M and q may not affect the team productivity, they do directly influence the effectiveness with which a project can monitor its progress and predict its future accomplishments. Generation of a WBS, of course, gives the parameter M . Monitoring the completion of milestones provides estimates for m and q . From these, projections of the end date and calculations for the accuracy of this prediction can be made. Based on such information, the project can then divert or reallocate resources to take corrective action, should progress not be deemed suitable.

A best straight-line fit through the cumulative milestone progress over the first r reports (of an expected $R = M/m$ reports) at ΔT intervals will predict the time T_M to reach the final milestone. It will also provide an estimate of m and q . The predicted completion date may be expected to deviate from the projected value (as a one-sigma event) by no more than [7]

$$\text{rms}(T_M) \leq 1.48(q/M)^{1/2}(R/r)^{1/2}T_M$$

within first-order effects. This bound permits the specification of WBS characteristics that enable accurate early predictions of future progress. The bound above shows that high accuracy depends on having a low q/M ratio. Therefore, *one must be willing to pay for WBS optimism at the cost of generating and monitoring progress relative to a more detailed WBS.*

As an example, suppose that a 10% end-date prediction accuracy is required by the end of the first quarter ($r/R = 0.25$) of a project. Then the milestone/optimism tradeoff factor according to the bound above is $M/q = 876$; hence, if the WBS is highly optimistic ($q = 1$), that WBS should contain 876 equi-duration milestones! If the project is confident that it can hold more closely to its optimistic schedule (has most contingencies provided for), with a $q = 0.25$, then it needs only about 220 milestones. A one-man-year project with bi-weekly reporting, one milestone per report (26 milestones in all), must demonstrate a $q = 0.03$ deviation factor! Less than one milestone, on the average, can be missed!

Perhaps this is a sad realization to some readers, but it is almost certainly a fact. The model for these figures may be held suspect, but let me say it is based on very conservative assumptions. It is, therefore, both necessary and important to generate a detailed WBS rather carefully, and to monitor milestone achievements relative to this WBS very faithfully, if accuracy in predicting the future progress of a project is of great importance.

11.2.2 Rules for Generating the Software Design Definition

Many of the rules to be applied in generating the architectural design are the same as those previously given in Section 11.1; therefore, I will not repeat them here. The reader should, however, apply them when entering upon the generation of an SDD, but orient them toward the criteria in Section 11.2, above. The reader should also consult Sections 12.1 and 12.2 in the next chapter for additional guidelines toward the architectural design of internal program procedures.

1. Complete the SDD, supplying the material indicated in the topical outline (Appendix D), documented subject to the rules in Section 11.4 and Chapter 16, according to the specific cases being documented.

Delete items in the outline that are irrelevant or that do not apply, or mark them as "not applicable." Mark purposely unspecified items as "later design prerogative," perhaps conditioned by the addition of modifiers such as "subject to approval." Cite existing material, such as Standards and Conventions, by reference only; include deviations from these, however. Figure 11-2 illustrates the format of the SDD.

2. Characterize the goals of the program development in terms of priorities among reliability, maintainability, modifiability, generality, usability, and efficiency.

3. Provide a top-down structured conceptual design that depicts the step-by-step decomposition of the program into its major modes, subfunctions, and algorithms. Document this design in a form suitable to later carry on the formal top-down detailed design (SSD phase).

In some instances, hand-drawn, properly annotated flowcharts or CRISP-PDL will suffice. In such cases, the program architects need not be overly concerned with the detailed correctness of procedures, so long as the basic architecture is sound enough for cost and schedule estimates. The flowcharts themselves are normally not a part of the SDD, but an overview of the architecture is.

4. Set work assignments and team interfaces. Choose these assignments so as to minimize the implications of human and program interfaces (see Chapter 10) and to maximize module inner cohesion (Chapter 4).

5. Design the major data structures and program data bases to the extent needed to establish the program architecture.

Specifically, leave to the detailed specification phase all considerations that do not directly respond to the SRD or that do not impact cost and schedule in a significant manner.

6. Couple the development of the SDD to any concurrent efforts toward completing the SRD or toward development of the SSD.

7. Determine how requirements shall be met by design and how each of the design items shall be prioritized to accommodate requirements.

Specifically, identify requirement not being responded to, or conflicts or changes in priorities, schedules, etc. Figure 11-4 illustrates one such method of illustrating how requirements will be accommodated.

8. Determine refined cost and schedule estimates based on a Work Breakdown Structure of tasks using the architectural design as a model for the estimation. Give variances or tolerances for each item estimated, and combine these into overall tolerance estimates for the costs and schedule. If a tolerance goal has been given, refine the architecture and cost/schedule estimates until the goal is met.

Program being developed MARK III Deep Space Station Command Subsystem Software		1.2 MKIII-OSS-CPA 7-5-75 page 2 of 4	
Requirement	Priority/ Sched	Accommodation Method	Adjustments
• as per requirements • • 4.3.2.5 Initiate and verify proper command transmission to spacecraft • • Abstract of SRD requirement SRD Reference	• • A/76 • • •	• • Real-time, Interrupt-actuated software/MODCOMP-II control of Command Modulator Assembly. Four-man, dedicated team • •	• • none • • • • • required schedule and priority can be met

Figure 11-4. A form for demonstrating the accommodation of requirements and priority (shown at Architectural Design Review)

9. Limit the program architectural phase of activity to that level of detail required to fix costs and schedule. If additional architectural details are required, develop these as part of the formal design phase.

10. Orient the Work Breakdown Structure toward measurable milestones that can be reported upon completion. Account for the possibility that certain tasks will need to be repeated, and include contingencies as separate tasks when appropriate.

11.3 GENERATING THE SOFTWARE FUNCTIONAL SPECIFICATION

The functional specification activity is primarily concerned with describing the detailed end-to-end characteristics of the program, without too much concern over what goes on in the middle (except as may emerge due to conflicts between these definitions and what the concurrent design effort has been able to support). Generating a set of functional specifications is at least as demanding a task as designing the program they describe, and specifications are just as likely to be in error. Because specifications cannot be "run," there is a tendency to postpone writing them until later, when there is a program that can be run. For many

reasons such an approach is wrong, principal among which is that the specification, not the program, should define correctness.

Many of the rules for software functional definition resemble rules given in previous sections, and also resemble design rules to be given in the next chapter. That is the case, as I stated earlier, because there is a conformity in the methods, not because the products are the same.

The software functional definition occupies a part of the Software Specification Document (SSD), which I call the Software Functional Specification (SFS). It is purely technical in nature; it pertains wholly to the external characteristics of the program. For this reason, many may choose to treat the SFS as a first (or separate) volume of the SSD. I have shown the SFS as an integral part of the SSD, inasmuch as it may be influenced by the internal algorithms in concurrent design or coding activities.

The Software Functional Specification responds to, extends, refines, and then documents the technical concepts laid down in the SRD and SDD. The SFS is not a mere restatement of the SRD or the SDD—it *defines* the way the program shall respond in order to fulfill the requirement.

The SFS, when complete and approved, satisfies the following criterion:

It defines the meaning of program correctness; any program meeting the technical and documentation specifications will be deemed a satisfactory deliverable.

During the completion process, each phase of the SFS development satisfies the following criteria:

- a. It is sufficient to initiate the development of user manuals as a separate activity, parallel to (but coordinating with) any concurrent program development activities (design and production).
- b. It is adequate for continuing the program development activities (design and production) with reasonable assurance that major revisions will not be necessary.
- c. It is reviewable by project and user personnel on the basis of its technical feasibility and accuracy, in accordance with the SRD, SDD, and the other criteria above.

The rules for generating the remainder of the SSD (the Programming Specification) are contained in the next chapter.

11.3.1 Rules to Structure Functional Specifications

Program definition is primarily concerned with deciding *what* the external characteristics of a program shall be, and leaving *how* these

functions are to be implemented to the program design activity. The guidelines in this section aid in identifying the functions and developing the relationships between the functions and the data they process.

1. Complete the SFS portion of the SSD as specified in the topical outline supplied in Appendix E, documented subject to the rules in Section 11.4.

Segment the material into semantically refined, hierarchic levels of detail. Cite existing material by reference only if it is a stable, maintained documentation. Figure 11-5 shows a graphical outline of the SFS within the SSD.

2. Start the hierarchic development with a one-page block diagram and accompanying narrative that shows the program properly imbedded in its environment and defines the major system and data interfaces.

3. State and display processing specifications primarily as functional "black box" transformations of input data and input conditions into output data and output conditions.

4. Display the distinct program modes, preferably using data-transformation descriptions in the form of information flow diagrams and explanatory narrative. State the logical conditions that invoke and terminate each such mode. Define intricate mode selection and transition logic, preferably in the form of decision tables based on program control inputs.

5. Define functional criteria for the program in terms of specifications of input data, processing, and output data, and by criteria that define ranges of acceptable measured behavior (see Chapters 14 and 15).

6. Describe input and output data sets in terms of their information content, the substructure imposed by relationships between items, and the correlations among data items. Leave to later design the details of data structuring to accommodate accessibility and storage, unless those considerations are already in effect (existing data bases), or are necessary to state the program functional definition.

7. Strive to structure specifications into modular forms that have as few connections as possible with other specifications not in the same direct line of hierarchy. A connection in this sense refers to dependencies and interrelationships between specifications. Well-modularized specifications can lead to well-modularized designs. As aids toward this end, be guided by the following:

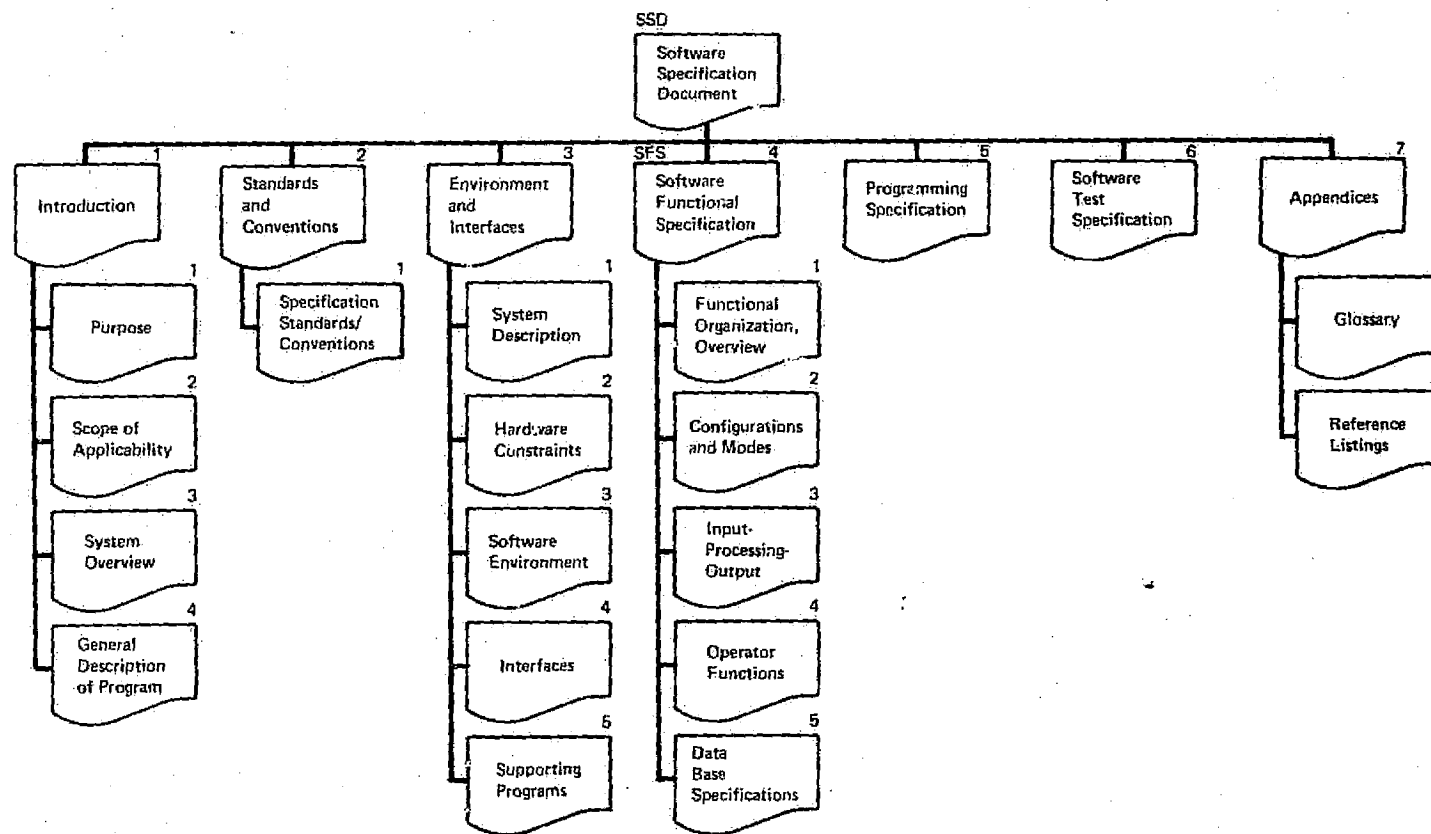


Figure 11-5. Graphical outline of the SSD with functional specification items emphasized

- a. If two specifications correlate, attempt to make this connection take the form of a shared subspecification, i.e., a common definition.
 - b. If a specification of a part of the program's behavior at a given hierarchic level depends on logical input conditions, then strive to partition that specification into subspecifications that will be independent of those logical conditions at the next hierarchic level.
 - c. Avoid specifications in which logical control is to be specified by a subdefinition at a later hierarchic level. That is, impose control definitions downward through the specification hierarchy, subject to (b), above.
 - d. Strive for *functional* specifications, in which all elements of each are related to the performance of subfunctions that combine to achieve a single, nonvarying functional goal. When practical, state subspecifications so as not to imply strict logical sequence of operations so much as subfunctional composition and precedence.
 - e. Avoid specifications, other things being equal, whose elements modularize unrelated actions together (i.e., which specify a set of subfunctions that do not have a common functional goal). A later alteration to such a definition could generate profound side effects in other definitions.
8. Extend the hierarchy of specification detail to that point for which the set of final definition "stubs" (leaves of the hierarchy tree) contains no subset of elements that could be useful as separate or common subdefinitions of other definitions. Each such definition element should represent a concept that is entire and within comprehension.
9. Coordinate functions that apply to major information structures so as to isolate dependencies on internal data-type, record format, etc., for each structure.

This will tend to minimize redesign and recoding, should the data representation structure change. Specifically, define operations that process information elements by field identifier or name, rather than by position. Describe operations in terms of the field or fields necessary for the operations to accomplish their functions.

11.3.2 Rules for Enhancing Program Utility

1. Specify the program so as to allow flexibility in tolerances and formats, and to adjust criteria for control decisions without changing the basic structure.

2. Specify attributes of the program in such a way that the program can run, at user option, in a degraded fashion when input data quality has degenerated.

Specify provisions, in such cases, to print out an alarm and continue, except, perhaps, in interactive environments where a stop and go-ahead command can be accommodated. In any event, retain the data and work up to the halt point.

3. Specify program responses for recognized, expected modes of failure so as to "fail soft" or "fail safe," according to the cost or risk of the failure.

4. Orient the computer-to-human interfaces toward the user and/or operator.

Such human engineering can add much appeal to the program, just as an interior decorator can increase the appeal of a home. However, base such interface definitions primarily on requirements, projected performance gains, and implementation costs, and secondarily upon aesthetic appeal.

5. Use descriptive messages to describe errors sensed during operations whenever storage permits.

Avoid messages which refer to conditions that violate the program's error criteria as being "illegal" (against the law), when what is meant is "improper," "invalid," or "not permitted" (in violation of the program's capability).

6. Specify program characteristics that foster testability; in some cases, a program function can be defined in terms of the tests that validate correctness. Include, where practical, self-diagnostic capabilities, such as data checking and consistency monitors, into the functional definitions.

11.4 DOCUMENTING TECHNICAL REQUIREMENTS AND FUNCTIONAL SPECIFICATIONS

Both technical requirements and functional specifications deal with program external characteristics, and, therefore, the documentation of each somewhat resembles the other. The rules in this section set forth standards for supplying graphic material and narrative to describe both.

It is worth remembering that the SRD and SDD are chiefly management-oriented, and are not particularly useful as surviving documents, except, perhaps, as a means to record project history. That is, their useful lifetimes do not generally extend beyond the software development period.

The SSD does survive, and contains the complete technical, as-built description of the program produced. The level of detail in the SSD is, therefore, much greater than in either of its antecedents. Further documentation rules for program specifications are found in the next chapter (Section 12.7), and Chapter 16 has definitions of standard classes of documentation. Also see Section 16.3.2 for documentor responsibilities.

11.4.1 Rules for Organizing Documentation

1. Document software items, such as requirements, definitions, etc., concurrently as those items are being developed.
2. Organize material in the documentation into hierarchic levels of increasing detail. Arrange the material in such a way that the upper levels are directly suitable for extraction and presentation at reviews.
3. Display management and planning information in a top-down format, in which the top level summarizes management resource requirements, project duration, etc. Detail subsequent levels into categories that show increasing levels of rationale for items appearing at previous hierarchic levels.
4. Organize technical material in Dewey-decimal order of hierarchic definition detail as illustrated in Figure 11-6. The numbers for this hierarchy may be obtained as follows:
 - a. If s is the Dewey-decimal identifier of a section in which an item numbered n (by rules in following sections) appears for further detailed expansion, and if this item has not previously appeared in the documentation hierarchy, then number the section detailing this item as section $s.n$ of the document.
 - b. If an item number m has been referred to as a common item in several places in the document, its description should appear in some separate section, say, section number d . In this case, append any new information learned about the common item in section s to section d of the document, and annotate section s , item m , with the fact that full information about this item appears in section d .
 - c. Start the top-level Dewey-decimal identifier d for such common items with alphanumeric level-1 section number for ease in locating and distinguishing the documentation section number.

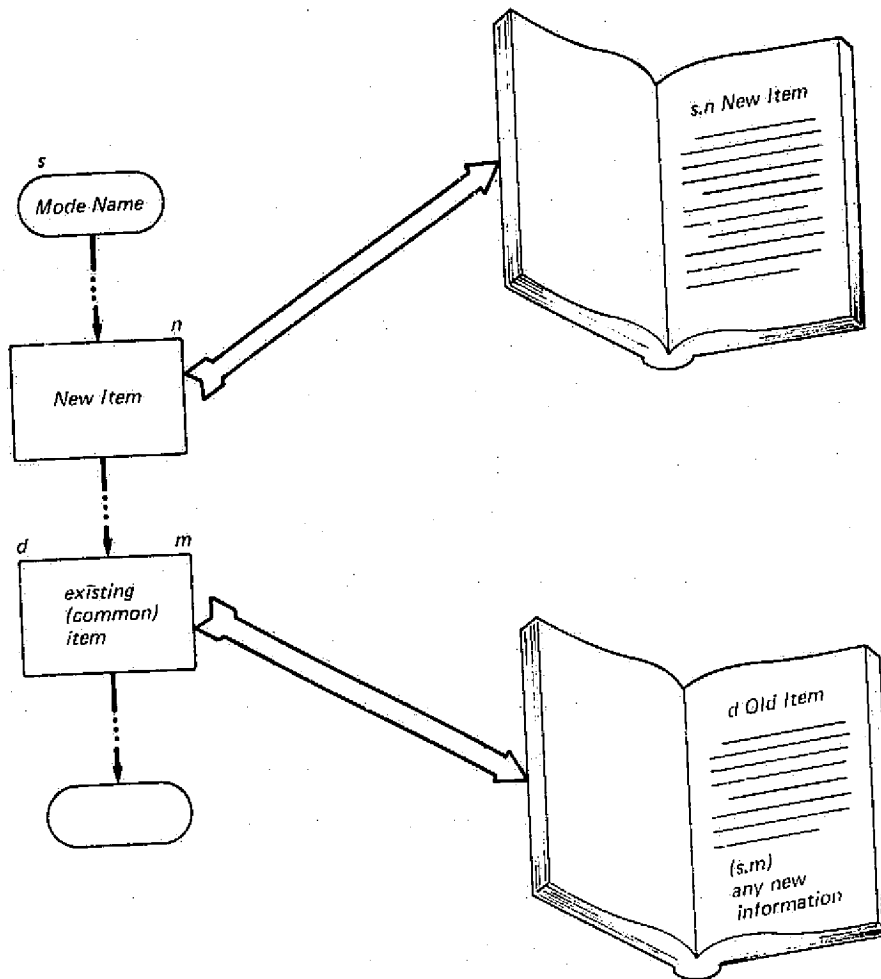


Figure 11-6. Organization of new design items and collection of subspecifications into the SSD

- d. Denote the n th table appearing under action number s using the descriptor $s.T_n$.

11.4.2 Rules for Representing Functional Information

1. State and display processing requirements in the form of "black-box" modes of operation, and state the general logical conditions that invoke each operational mode (see Figure 11-7).

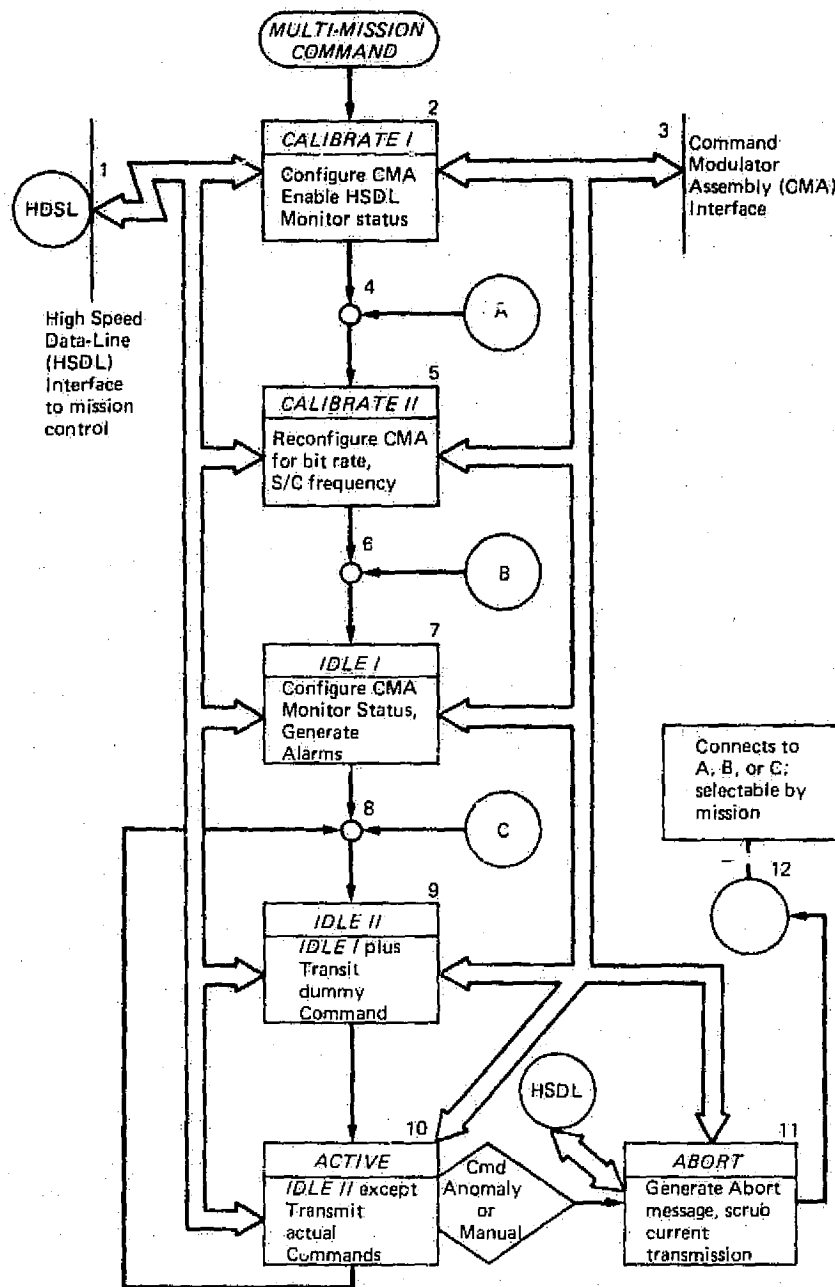


Figure 11-7. Input-processing-output modes of operation for the software in a multi-mission ground command system for spacecraft use (accompanying narrative to this mode diagram will detail functions and events actuating transfers between modes; this is *not* the program flowchart from which coding proceeds)

2. Display major processing functions within each mode primarily as a top-down hierarchy of input-processing-output charts (see Figure 11-8) and accompanying narrative, rather than as control-logic flowcharts, unless such charts are necessary to display the logical sequencing of program requirements.

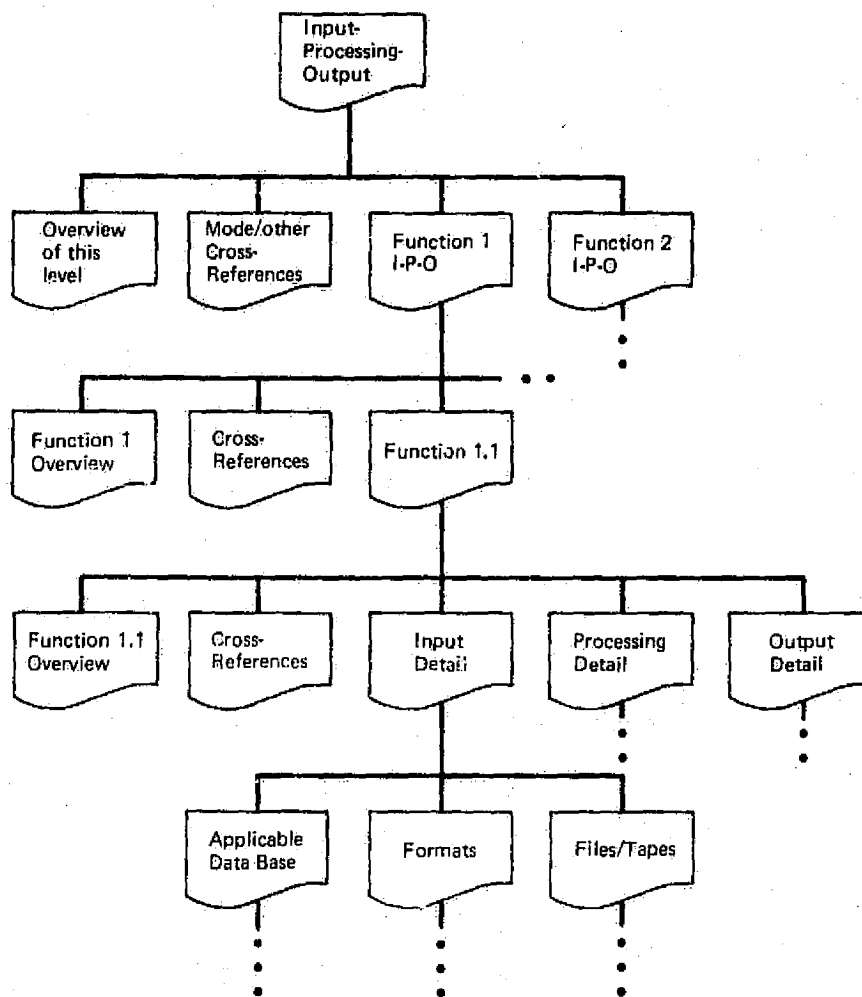


Figure 11-8. A sample for representing hierarchic input-processing-output (HIPO) functions in documentation (two methods of detailing are shown, and others may also be used; cross-references may thus be necessary between "cousin" sections of the document)

3. Use information-stream diagrams to identify the source of each required input, the required routing through processes, and the destination of each required output.

For example, inputs may emanate from existing data-base files, or from information to be supplied by a group of users or operators. Outputs may be destined to files, or, perhaps, to users by mail.

4. Describe inputs and outputs as (perhaps formatted) information structures, rather than data structures or storage structures later to be designed, unless these structures represent existing data bases, or else are relevant to the satisfaction of a stated software requirement.

5. Document complex logical multi-mode requirements in the form of decision tables. A mode is defined as a way of operating a program to perform a certain subset of the processing requirements that normally are associated together in the program function.

6. If internal data interfaces are required to define a program function, describe the pertinent assumed characteristics of the interfaces in supplementary data interface definition tables, hierarchically evolved. Keep the state of all such interfaces current, and make all references to data interfaces during the definition activity agree with the tables.

7. When it is necessary to define functions algorithmically, use CRISP-PDL or some similar syntactic structures superimposed on (abbreviated) English, mathematical or programming terminology to describe such algorithms, or produce flowcharts and narratives along the guidelines given in the next chapter (see also Chapter 17).

8. Use full graphic aids, such as mode diagrams, information (or data) flow diagrams, data structure diagrams, decision tables, timing analyses, and (only when sequence is important) control-logic flowcharts, along with accompanying narrative. Such accompanying narrative should supply explanations and detail not contained in the graphic aid (such as rationale or instructions on how to interpret the graphic). Do not leave graphics unexplained in the narrative.

11.4.3 Rules for Graphic Representations

1. Limit all graphic representations, such as mode diagrams, decision tables, information flow diagrams, data structure illustrations, and control-logic flowcharts, to one page each, except in unusual circumstances. Use hierarchic expansion and nesting to provide detail, rather than off-page connectors.

Figures 11-9 and 11-10 illustrate a hierarchic definition of program modes.

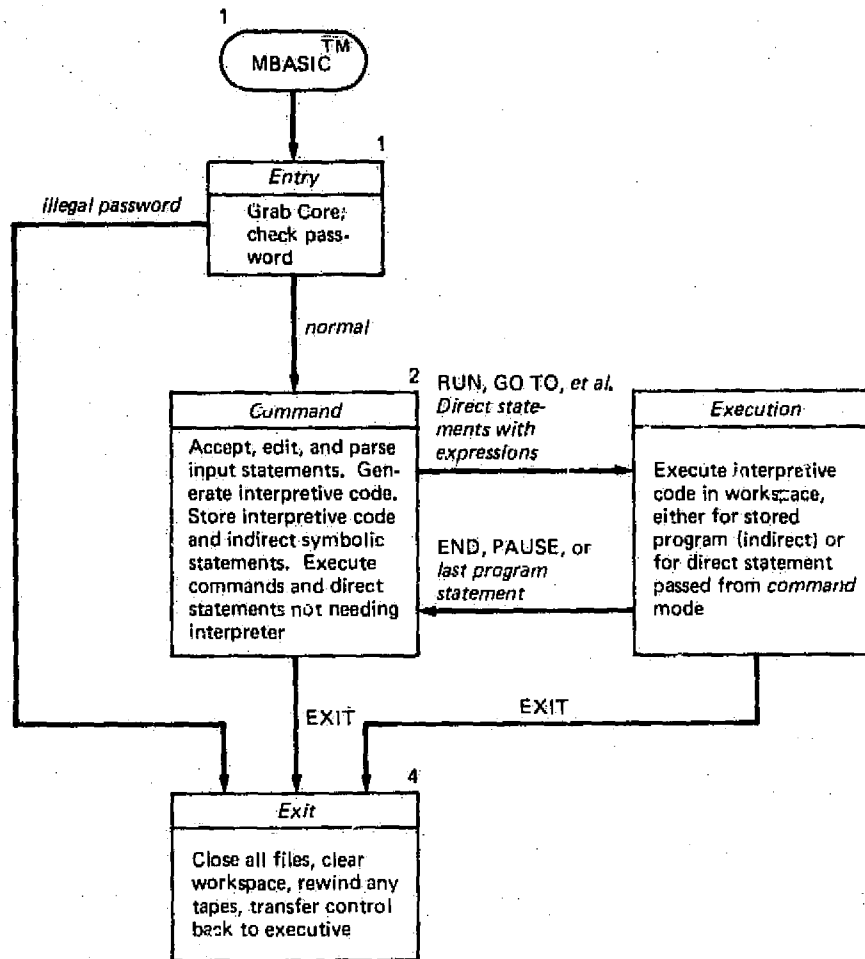


Figure 11-9. A simplified mode diagram for the MBASIC™* interactive language processor (boxes show required modes of operation and reasons for transitions between modes as stated in the language specification; this diagram does not show the design of the program into modules)

* A trademark of the California Institute of Technology.

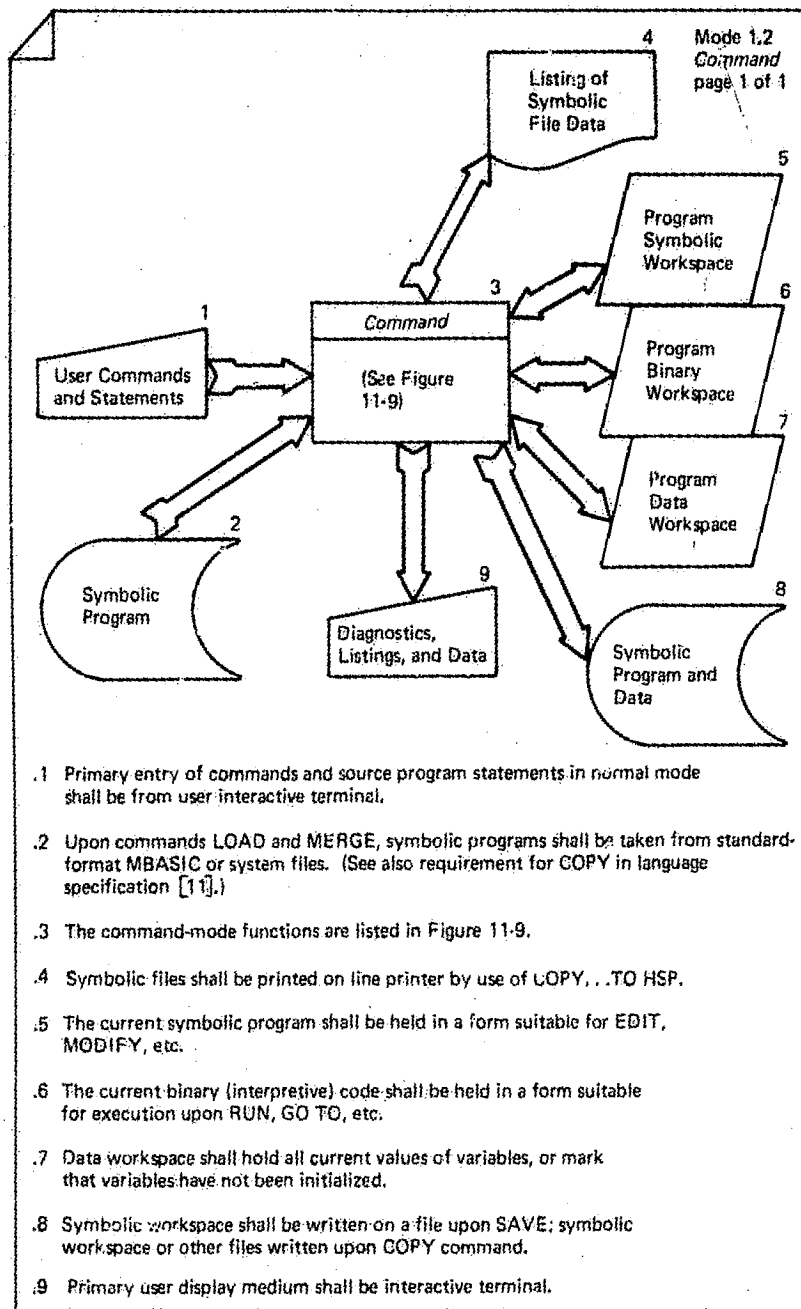


Figure 11-10. A brief multi-mode information flow diagram and narrative that lists a few of the top-level requirements for the command mode of an interactive MBASIC™ language processor (later levels refine the separate operational mode requirements within the command mode).

2. Conform symbolic graphical representations of I/O media and processing operations to ANSI-X3.5-1970 Standard Flowcharting symbols (Appendix B).

3. Draw information-flow lines on charts as double-line arrows, preferably having both heads and tails in the direction of flow. Draw control-logic flow (sequence) lines as single solid lines terminated by an arrowhead, only one arrowhead per line segment. Connect comments to charted symbols by dashed lines without arrowheads.

These conventions are summarized in Figure 11-11. Figure 11-12 illustrates an input-processing-output definition table. Detailed flowcharting standards appear in Section 12.7.2.

4. Distinguish processing functions on hierarchically expanded charts by horizontally striping the chart symbol. Use vertical stripes to indicate process definitions detailed elsewhere. Use horizontally striped symbols to denote hierarchically expanding information structures within the current document. Use the appropriate ANSI standard symbol (Figure 11-13).

5. Number all boxes, at the top right of each, on all charts. Use a pre-order traverse of nodes (Section 5.1.3.3) for procedural flowcharts and topological sorting (Section 4.3.3) for information-flowcharts. Conform narrative descriptions to the chart numbering scheme, such as illustrated.

6. Annotate input and output data bases with appropriate information to locate the complete data-base interface documentation.

7. Label boxes on functional diagrams with the function they perform, letting the shape of the box define the agency by which that function is accomplished.

For example, raw input data may be indicated as being recorded on a disk for later backup. The box, however, should not be labeled merely "disk," but with its function, such as "digital original data record."

11.4.4 Rules for Naming and Referencing

1. Develop a hierarchic table of contents for the technical descriptions using the numbering scheme in Rule 5 of 11.4.3, above, when applicable (such as flowcharts—see Section 12.7.2). Use the upper left of each box in charts to refer or cross-reference the next layer of detail through this table of contents.

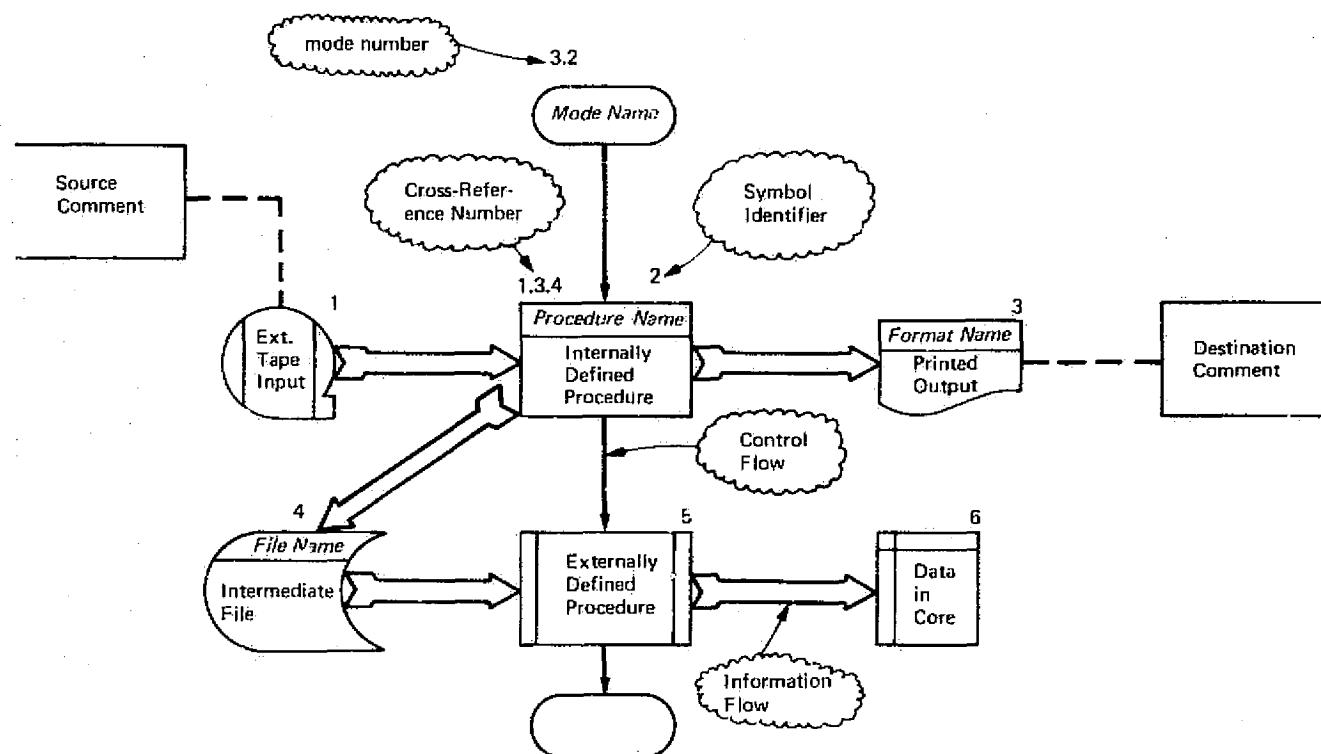


Figure 11-11. Information-flow diagram standards (annotations in "clouds" are not part of standards)

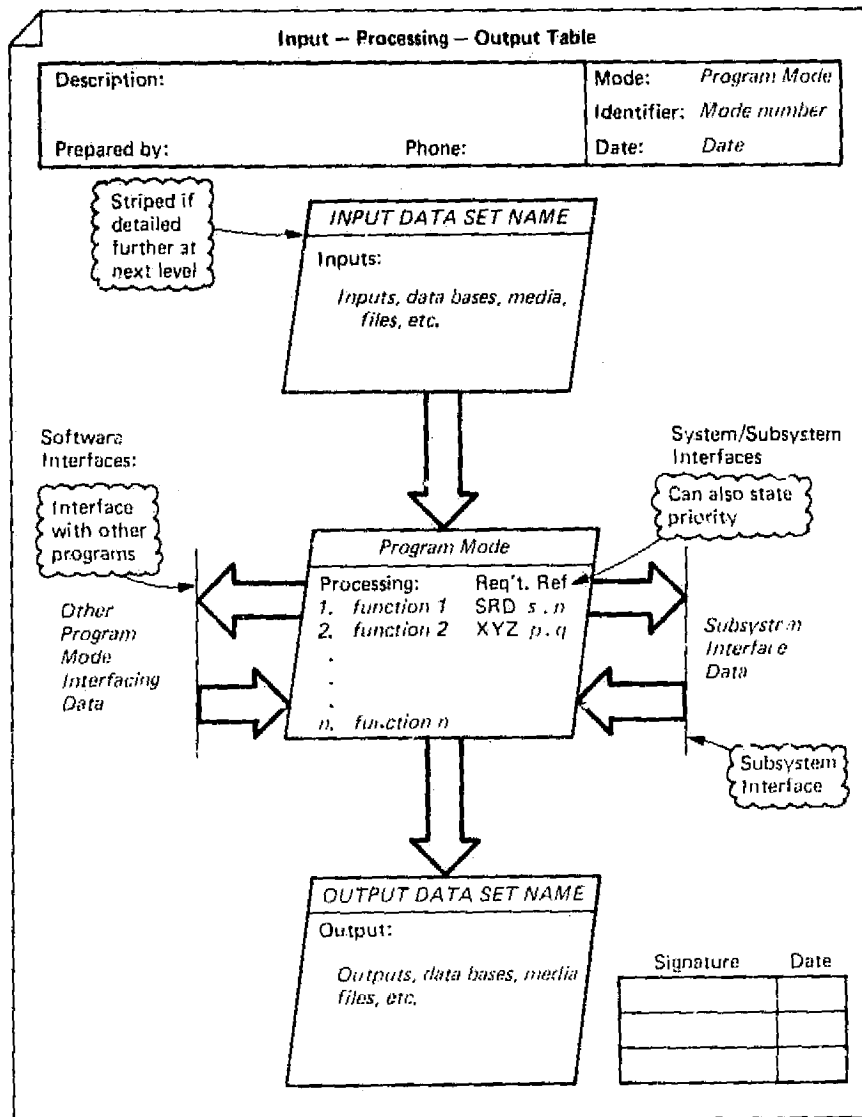


Figure 11-12. Input-processing-output definition table, showing interfaces to other program modes or other subsystems

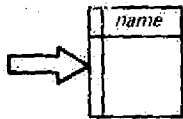
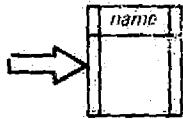
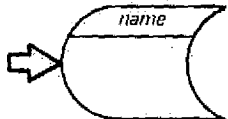

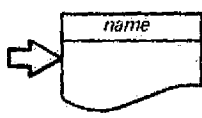
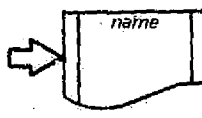
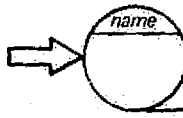

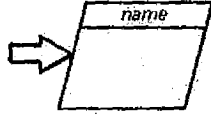
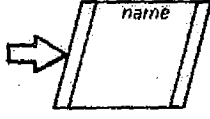
Structure type	Expanded elsewhere in this document	Not expanded here, but documented elsewhere (reference attached)
in core		
online storage		
document		
magnetic tape		
arbitrary medium		

Figure 11-13. Striping conventions for information, data, and storage structures (unstriped symbols are not expanded elsewhere)

2. Name all processes and information structures used in later references using mnemonically derived symbols. Insert all such names in the document glossary.

Remember that names given to required modes of operation or to processing requirements in the SRD are not necessarily going to be the same as the names of processing modes or actions defined in the SDD, and these in turn may be altogether different than the names of executable modules and data structures specified in the SSD.

3. State the mnemonic derivation of all names used in a description unless such names have previously occurred in an ancestral definition of the hierarchy. Names appearing in "cousin" descriptions, unreferenced in their common ancestor, should be defined mnemonically in both cousins. Insert all mnemonic derivations into the glossary.

11.4.5 Rules for Decision Table Formats

1. Format decision tables into the standard form used in Chapter 8 and illustrated in Figure 11-14. Fill in these tables according to the remaining rules in this section.

2. Separate Condition/Action and Stub/Entry sections by double or boldface lines.

3. Limit the size of tables to one page using hierarchic subtables if necessary.

4. Designate tables by their mnemonic name and assigned Dewey-decimal number at the head of the table.

5. Normally, strive to limit each table to no more than 6 conditions, 15 decision rules, and 15 actions by hierarchic nesting of table entries.

6. Use dashes to indicate immaterial condition entries and ignored actions in a rule. Do not leave blank entries.

7. Use a consistent set of indicators in LEDT condition entries ("Y" or "N," "T" or "F"). Use "X" in action entries to indicate single actions or multiple actions where sequence is immaterial. If action-item sequence is material, number such items in the action entry in their order of logical precedence. Assign equal numbers to processes having equal precedence.

8. Arrange the action-stub items in their general order of execution, if possible. Number each action item and indicate whether there is further hierarchic development.

One-Page Limit

Decision Table

Description:		Title: <i>TITLE</i>	
		Identifier: <i>TABLE s. Tn</i>	
Prepared by:		Date: <i>date</i>	
		Phone:	

Conditions	Rule	1	2	3	4	5	6	7	8	9	10	11	12	13	14	ELSE	COST
	Prob:						.05										
1.							Y										
2.							N										
3.							N										
4.							—										
5.							—										

Actions	Rule	1	2	3	4	5	6	7	8	9	10	11	12	13	14	ELSE
1.																
2.																
3/D5.							X									
4.							1									
5/.							X									

Sequential Test Procedure:

signature	date	signature	date
-----------	------	-----------	------

Figure 11-14. Summary of decision table documentation rules

For example, if there is no further detail, merely use the action number n ; if there is more, but the action is not one common to other charts, then use $n/$; if the action is one common to other charts, described by the cross-reference x , then write n/x .

9. If decision rules invoking the same action have been combined, then state, if possible, the relative frequencies of the uncombined rules if there is to be an attempt at optimizing the decision logic.

10. Enter only rules that correspond to true alternatives into the table. If the order of rule testing is necessary or pertinent to specify this entry, then state the sequential testing procedure or give a reference to the procedure elsewhere in the documentation.

11. Unless otherwise stated, assume that control flow of each table will merge into a single (proper) exit at the end of the table.

11.5 RULES FOR THE SOFTWARE DEVELOPMENT LIBRARY

1. Begin with the full document outline and enter an asterisk (*) in front of all sections to be completed. As sections are subsequently completed, remove the asterisk. If a section expands and creates new subsections, enter headings for these into the Table of Contents (prefixed with an asterisk, if not complete).

2. Start the description of each expanding subsection unit on a separate page. Affix to each page a block in the upper right-hand corner containing the date, "page__of__," the subsection mnemonic name, and Dewey-decimal number of the unit. Date each page with its date of submission into the Software Development Library (SDL). Any changes should require redating that page and reapproval, if the unit is in project change control.

3. Affix a signature control block to each unit to contain the initials of the person writing it and the concurring initials of the project manager.

These initials testify that the documentation specifies the desired program behavior, that it is being accepted into project change control, and that any further considerations which must be based on that unit can begin.

4. Make the SDL the central repository for all documentation, be it working-level, look-ahead, or approved.

5. Maintain a glossary and data interface definition table definitions as alphabetical files in the Software Development Library (on 7-1/2 × 12-1/2 cm (3 × 5 in.) or IBM cards, for example, if not in computer files), to allow for flexibility in refinement of information.

11.6 SUMMARY

This chapter has given a set of guidelines for developing and documenting functional characteristics of a program, estimating costs and schedules for its production, setting design criteria, establishing the program architecture, providing management visibility and controls, and enhancing program utility. These guidelines become standards when it is necessary to have a commonality of approach, or when these guidelines contribute to the goals of the software project (e.g., minimize life cycle costs, maximize income or utility, reduce cost or schedule overruns, etc.). Guidelines in the next chapter extend the standardized approach to software into the design arena; subsequent chapters then cover compatible standards for coding, testing, and quality assurance.

XII. PROGRAM DESIGN AND SPECIFICATION STANDARDS

As I said in Chapter 4, there is probably little hope of ever standardizing the design process itself. However, much progress in this direction can be made by adopting disciplines that encourage the identification of goals, problem constraints, design parameters, and solution alternatives, and that coordinate these into a uniform, readable product. The design documentation should describe the program to the extent the program can be understood without reference to the listings.

The guidelines that follow in this chapter summarize a set of standards and practices which encourage design as an activity separate from, but coordinating with, later coding and testing. The rules also integrate the program design in a like manner with the functional specification activity addressed in Chapter 11.

12.1 RULES FOR STRUCTURAL DESIGN

There is a difference between designing the structure, or architecture, of a program and designing a program in a structured way. Program structure refers to the way in which a complex algorithm may be characterized in terms of successively simpler forms. The foremost thing to remember while attempting to design the program structure is that it is the *structural framework* that is being designed—not the procedure, not the data formats, nor the control logic.

The structure of a program primarily manifests itself in the selection among alternatives in various design categories, as described in Section 2.2. Structure deals with relationships such as cohesiveness and coupling within and among the program submodules, the architecture of the functions and data flows, the intended operational modes, and the conceptual information structures. Structural design forms the basis for the remaining detailed design effort.

During the early parts of the design phase, there is a strong, almost irrepressible urge, to begin the detailed procedural design before the program overall architecture has been established. Such techniques as structured flowcharts and CRISP-PDL may seem natural tools for defining structure, but let the designer be aware that such tools only design procedure. If one begins a procedural design and finds difficulty with data structure definitions, resource allocations, functional definitions, and the like, then probably either the program architecture or the functional specifications are not yet sufficiently well understood to proceed further. Figure 12-1 illustrates that processing algorithms are but one area among design considerations typically imbedded within the total design task. The considerations which interrelate with the processing design process demand that a proper foundation for such design be established. Larry Hawley, a colleague at JPL, remarked in this respect, "No amount of riveting can strengthen a building enough to make it a skyscraper if its foundation and structural members are unsound."

The guidelines that follow in this section reflect software development practices which help to establish a sound program structure.

1. Establish design tradeoff criteria at the outset. Make tradeoff analyses of all potentially useful or competing methods for implementing critical parts of the design. Don't throw these away. Put them in the Project Notebook.

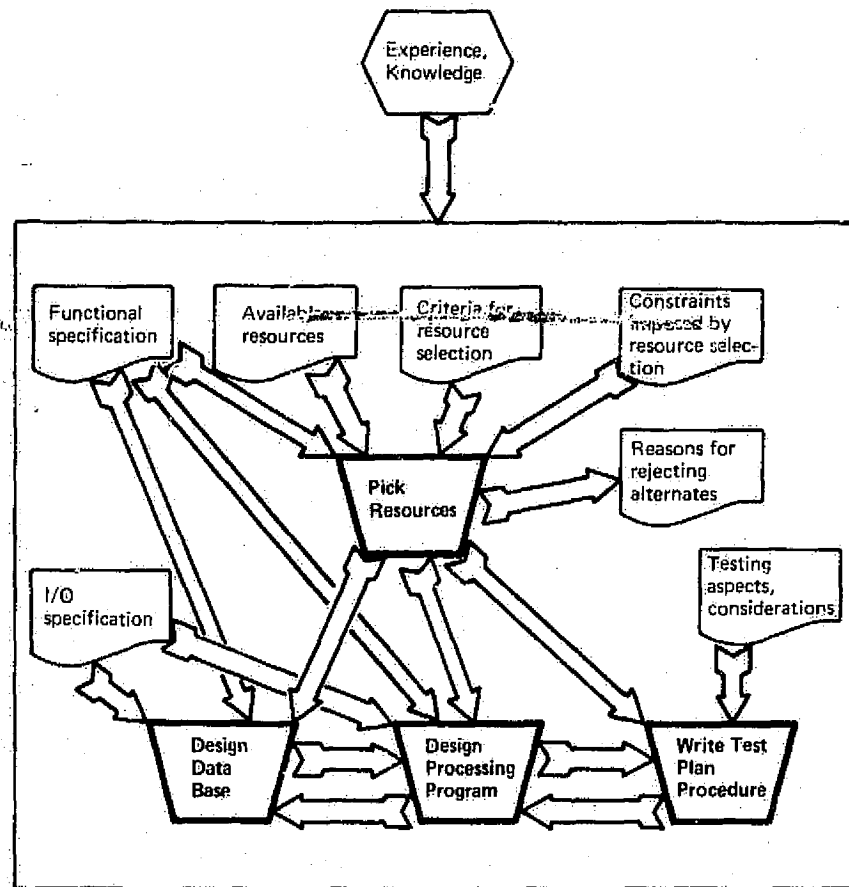


Figure 12-1. The Software Design Task and some conceptual flows of information

2. Begin the major program architecture as a one-page block diagram of functions, such as Figure 12-2, showing major information flows, data bases, and data structures.

3. Obtain a firm commitment from cognizant individuals over external data bases to maintain a stated interface structure, i.e., format, content, and integrity; to inform your project about the extent, form, and schedules of any upcoming maintenance needed on the bases; and to consult with your project before any subsequent changes to the committed interface are made.

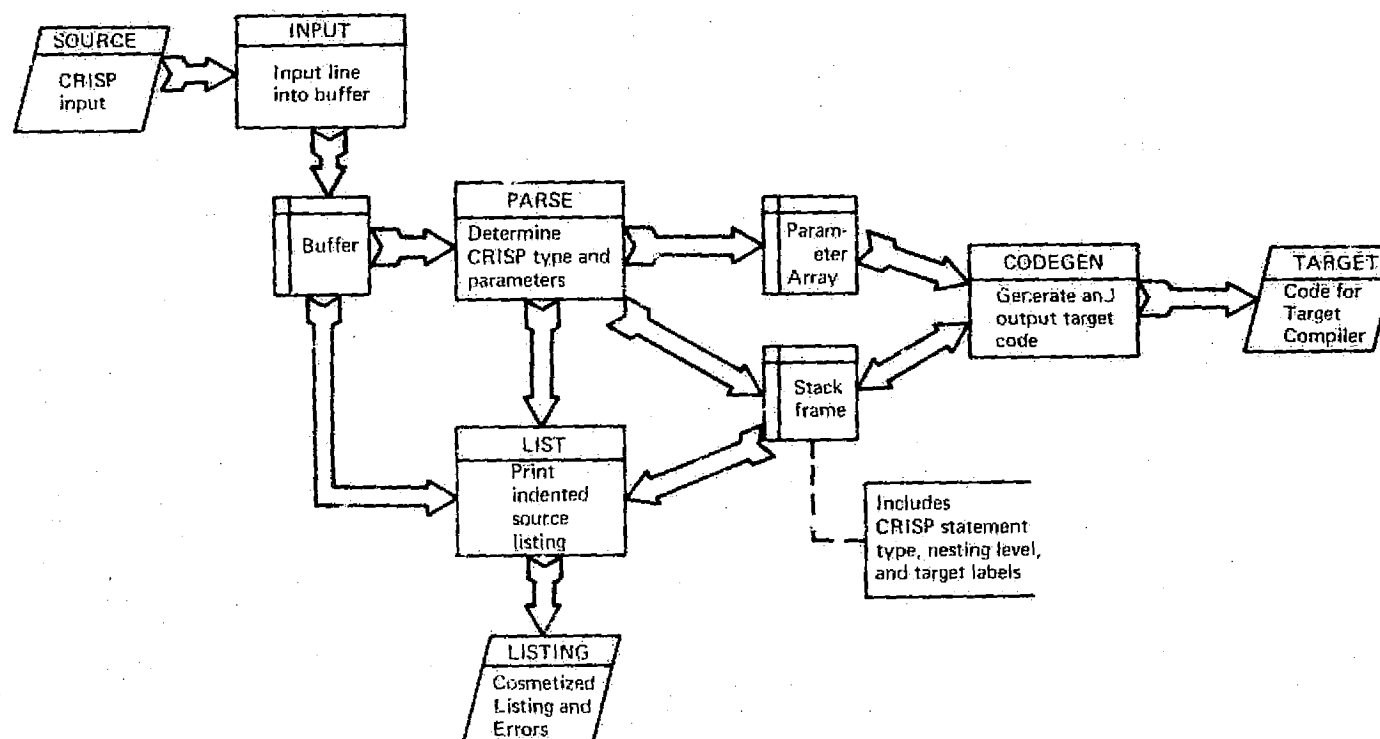


Figure 12-2. Top-level block diagram of a conceptual CRISP preprocessor, showing data flow, major operations and data structure, inputs and outputs

4. Analyze functional behavior, data structuring, and coupling modes of modules, and strive to maximize module cohesion and minimize module coupling, as described in Section 4.4. In particular, strive to:

- a. Define levels of access for all large or potentially changeable data structures or other resources, to "hide" the actual data or resource behind functions which access the data in the prescribed way. Pass information to and from these structures as arguments.
- b. Name modules using single-verb, single-specific-direct-object actions that truly reflect the intended function of the module. This promotes functional cohesion of modules.

5. Use top-down, hierarchic, modularizing, structured programming techniques that refine the program procedural design into a hierarchy of detailing decompositions of the program function into parts capable of being perceived as a unit. Maintain consistency and concurrency between data-structure and resource-access definition hierarchies and the program procedural-design hierarchy.

6. Use informal design look-ahead analyses to develop architecture, to study feasibility of ideas, and to identify common "basic services" that can be incorporated as subroutines before there is a large investment in formal documentation or code. Submit approved (signed-off) design items into project control for later coding.

7. Describe the relationships between functional modules and the data they access by stating how the *representations* of information, *manipulated* by functions, are to be *interpreted* by the "real world."

8. Develop decision tables for logically intricate parts of the program, to identify all relevant conditions and to define corresponding actions.

9. Study the programming language(s) to be used for implementation. Whereas the upper levels of design are encouraged to be as language-independent as possible, it is imperative that the designer be intimately acquainted with the target language and system capabilities, as they most certainly will impact the design.

12.2 RULES FOR DATA STRUCTURING AND RESOURCE ACCESS DESIGN

The fundamental problems in data structure design are deciding: (1) when to save the results of calculations, rather than recompute them; (2) how to store data items so that they are accessible by the operations which

will transform them into output; (3) how to organize the data so that storage and operations are efficient; and (4) how to document the data structure so that others can understand the operations performed on it. Such discussions are influenced by data type, amount of data, relationships among data elements, and the algorithms that operate on the structures.

The guidelines that follow primarily address the design procedures for structuring data; however, most of these apply equally to designing accesses to other resources, as well. I would like the reader to know that I intend that the same rules be applied in the more general case.

1. Start with the most abstract concept of the information structure of a problem. Then use stepwise refinements to create a hierarchy of more and more detail. At each step, refine the representation of data items or the operations on data items in a way which details, but does not conflict with, the previous levels of refinement.

Figure 12-3 depicts a top-level architectural tier-structure chart for a conceptual CRISP preprocessor. The indicated modules, data flows, and data structures show the designer's first concerns and decisions; later refinements delve into secondary details.

2. Concentrate primarily, at each level, with *what* is being done, rather than *how* it is to be done (which will be defined at later levels). It is proper to use a look-ahead design to check feasibility of the current level of thinking; however, submit only approved (signed-off) design items into project control and later coding.

A typical abstraction of a list structure for the PARSE module in a CRISP preprocessor might appear as depicted in Figures 12-4 and 12-5.

3. At each level of abstraction, attempt to discover only the relevant aspects of the information structures of the problem, such as:

- a. Amount and source of each information stream.
- b. Types and other attributes of data elements, e.g., units.
- c. Relationships among data elements.
- d. Composition of data constituents from data elements.
- e. Operations to be performed on the information.
- f. Frequency of access and response time.

Then invent conceptual data structures or refinements of previously defined structures which can accommodate (a) through (d), above. Invent

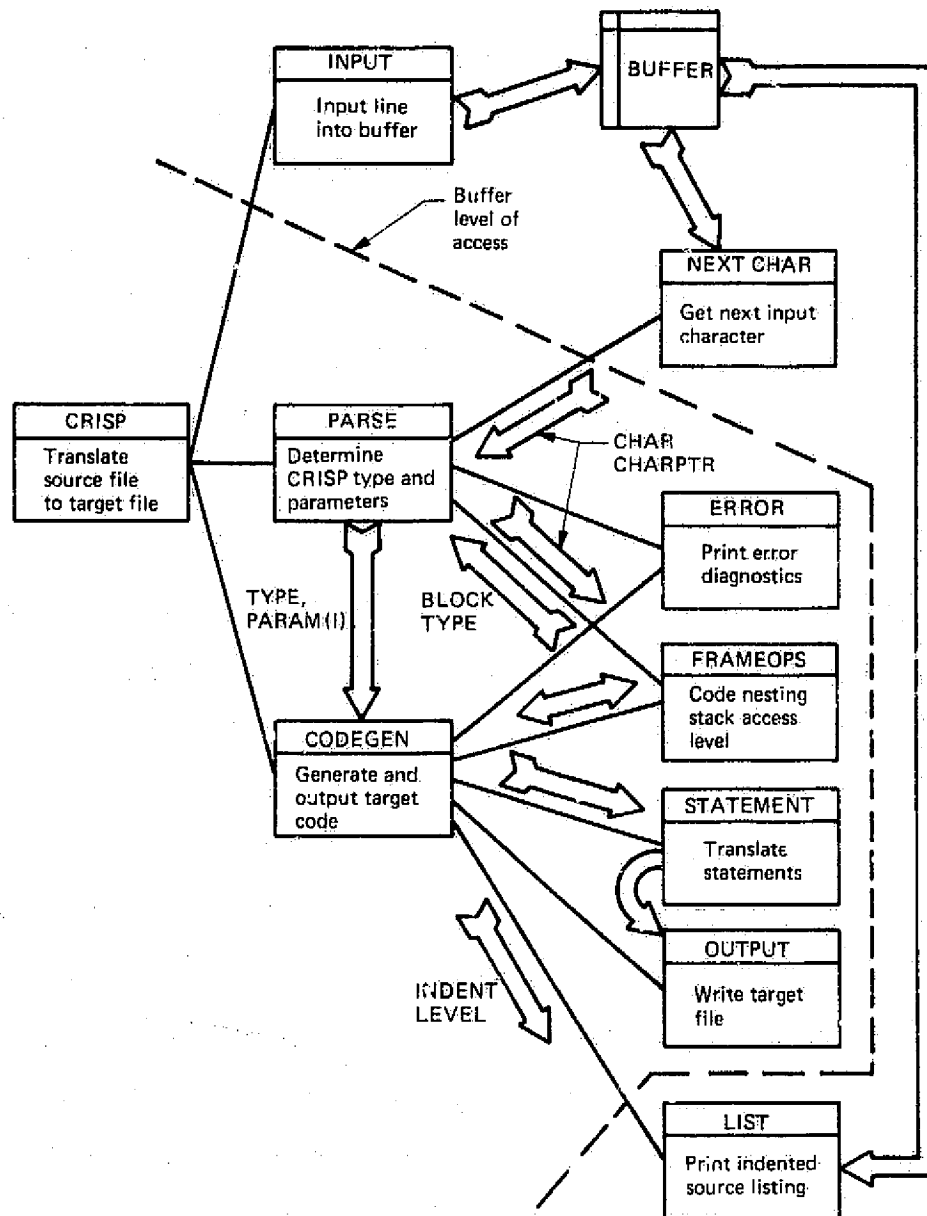
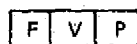


Figure 12-3. An architectural tier design chart for a simple CRISP preprocessor (note that some flows are indicated but not annotated with specific type or label information; other flows, such as communication with ERROR, have been omitted altogether, until a later abstraction)

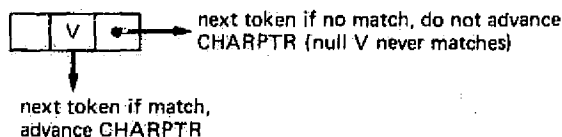
(a) 3-Field Tokens:



F = flag field
V = value field
P = pointer field

(b) Sequence:

If character at CHARPTR in BUFFER matches character in V-field of current token, next token is the one just below current token. If there is no match, next token is found by following P field:



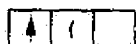
(c) Flag-Field Directives:



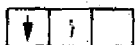
Set next parameter token; V-field = 0, P-field = CHARPTR. Set parenthesis counter PARENS = 1.



Increment V-field of parameter token, Advance CHARPTR, and advance to next token.



Increment PARENS by 1 and increment V field of parameter token by 1 if there is a match. Advance to next token.



If there is a match, advance CHARPTR and decrement PARENS by 1. Then, if PARENS = 0, advance to token below. If PARENS > 0, increment V-field of parameter token, advance to token in P-field, and advance CHARPTR. If no match, follow (b), above.

(d) Parameter Tokens: Contains length L and pointer to first character in the input buffer for each parameter string of CRISP statement:

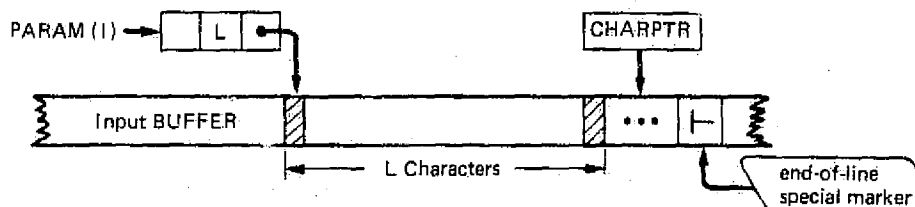


Figure 12-4. Design preliminaries for list tokens and conventions for list parsing of CRISP statements

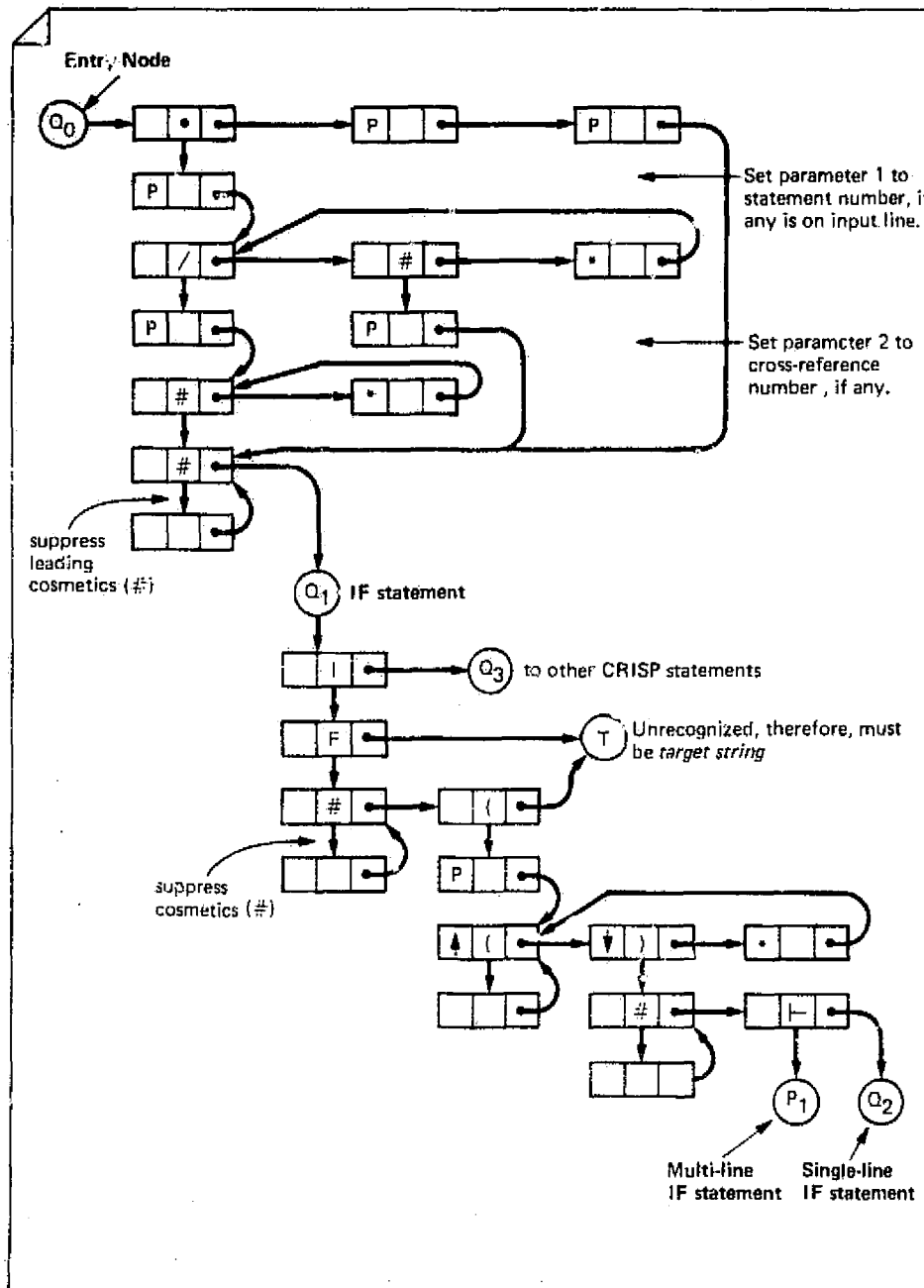


Figure 12-5. A partial list structure for the PARSE module in a CRISP preprocessor (logic for traversing the list tokens appears in Figure 12-4; node P_1 is the accepting state for recognizing the multi-line conditional statement)

conceptual operations α : these structures to accommodate (e). Assess the correctness of the representation; that is, convince yourself that operations on the data structures correspond to intended operations on the information structures (i.e., the real world).

4. Isolate data structures having the same operations and type attributes into a level of access. Make each level of access have a set of data resources which it owns exclusively, and within which only subordinate levels of access are permitted to access differently (see Figure 12-6).

5. Envision data items and the relationships among items as a "data graph" structure. Strive to keep such structures simple enough so that a two-dimensional one-page display of the data format with interconnections and indicators of subgraphs can be drawn to document the design of that structure. Whenever a data graph has more than one disjoint connected subgraph, consider whether the disjoint components might not be better defined as separate structures.

Note that table-driven algorithms simulating finite state machines are actually interpreters, in which the "program" is the tabular data structure. Selection of the next state is

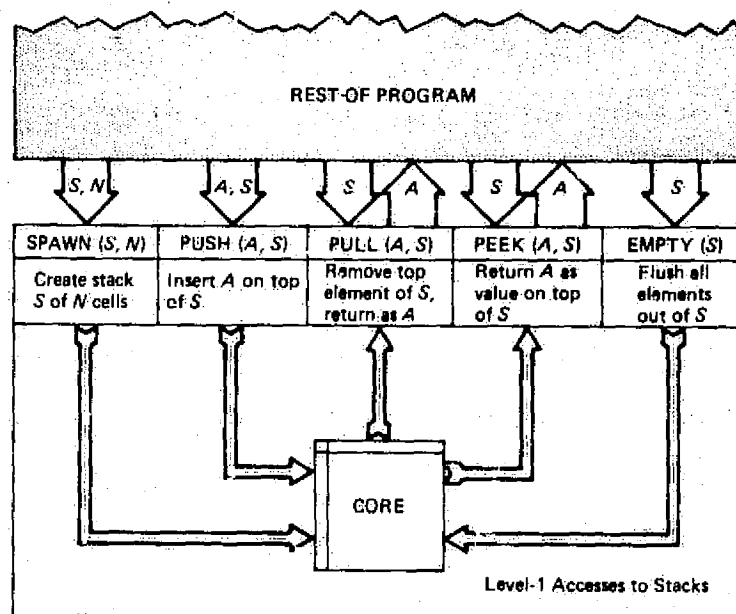


Figure 12-6. Organization of program modules into a level of access composed of a set of functions that own a data structure

analogous to IF or CASE, and cycles in the data graph are loops in the "program." The data graph, in fact, is the "program" flowchart. If this flowchart is unstructured, then that data structure suffers from the same faults as an ordinary unstructured program. The list displayed in Figure 12-5 is highly organized, but a few unstructured jumps out of loops exist, for example. The designer must assess whether the reasons for this lack of structure are justified with respect to the set of competing characteristics (Section 4.5).

6. Decide upon the degree of data packing to save space, as opposed to the lack of packing to save access time. If packing saves time as well as storage, consider whether the program will be more complicated or require additional expense (e.g., garbage collection).

7. Carry through look-ahead design efforts for each critical item all the way down to that level of detail required to ascertain storage, access time, and other considerations which could seriously affect the remainder of the data structuring. Perform such feasibility studies early enough so that the impact of any adverse findings can be accommodated.

8. For operations that act on data structures, specify operands by field name, rather than by an assumed substructure position. In this way, algorithms become insensitive to the substructuring of the data.

12.3 RULES FOR DEVELOPING STRUCTURED PROGRAMS

This section contains guidelines that relate to the formalized development of the detailed program procedure as a hierarchy of structured modules implemented from the top down. In principle, the coding can take place concurrently, also from the top down in execution sequence, after the architectural phase, as design items are completed.

The program architectural design phase may be considered to be ended when:

- a. All major levels of access have been identified, resources allocated, and the access-level-hierarchy established.
- b. The preliminary functional analysis and program structural definition exist and are documented in accordance with SDD standards.

- c. The level of detail extends only down to the point required to fix cost and schedule, within acceptable variances (say, a 10% goal).

The following rules outline the kinds of steps necessary to formulate and structure the program procedure in a disciplined way.

1. Define phase one of the program design as that amount of work which will be required to satisfy the architecture, or "design definition" requirements. This is a look-ahead effort prior to beginning the formal design-code-test cycle with concurrent documentation.

2. Do not begin the formal procedural design (the SSD) until the amount of work has at least reached that preliminary level which satisfies (a) through (c), above. Then begin the formal task of developing, documenting, and demonstrating this design *anew* at level one for the SSD.

3. Begin the main program, each common subroutine, and, perhaps, some of the major subprograms at documentation level one.

4. Monitor the program structural development using a WBS and tier chart, or equivalent. Each tier consists of all program modules that possess the same degree of hierarchic nesting in relation to the level-one main program.

Note that, in a program without subroutines or level-one subprograms, the tier number is the same as the documentation level number. The organization into tiers is illustrated in Figure 12-7.

5. Define phases of development subsequent to the preliminary (or architectural design) phase so as to agree with any given priorities or capabilities set forth in the Software Functional Requirements or in the Software Development Plan. Such phases are to be defined by sets of modules next in top-down order on subtrees of the tier chart (see Figure 12-8). The sets of modules for a phase are to be chosen as testable milestones that increase project productivity by demonstrating (using dummy stubs) the appropriate partial correctness of the program in responding to a stated functional requirement.

6. Orient the design process so as to complete an entire defined phase as a major project milestone. Complete and document all modules at the current formal design phase before proceeding to the next format phase. (This rule does not pertain to informal look-ahead efforts.)

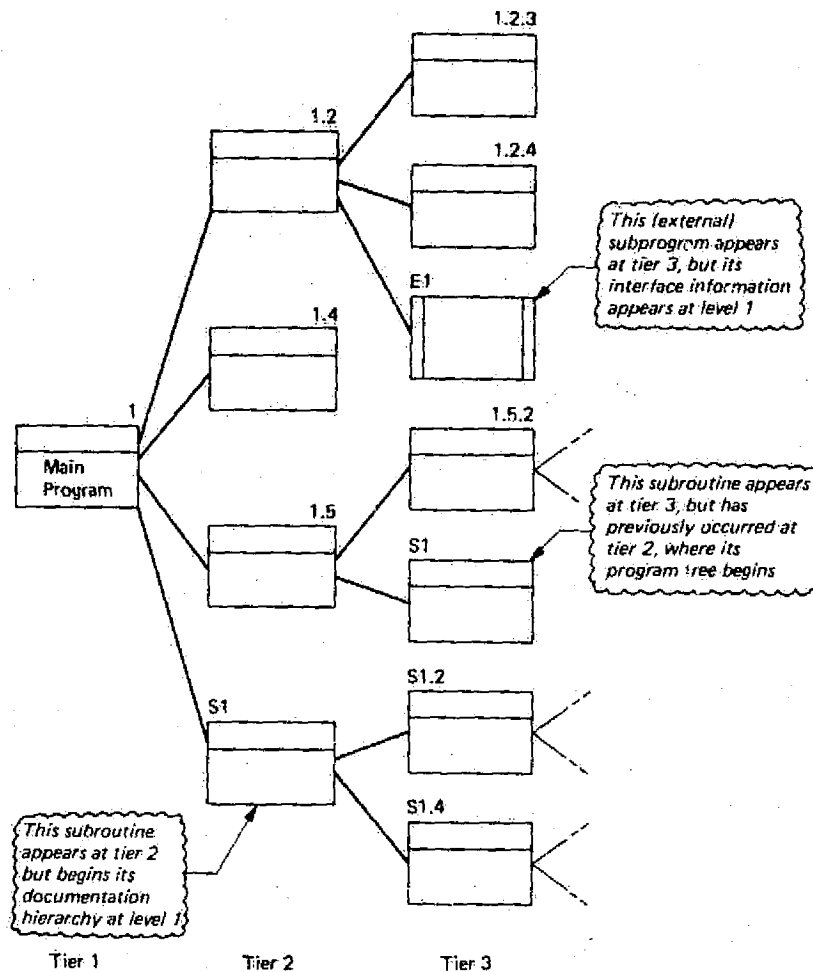


Figure 12-7. Organization of program modules into tiers by nesting hierarchy (level number refers to documentation hierarchy; tier number refers to degree of nesting within main program)

7. Direct the effort within defined project phases toward completion of modules having the lowest-numbered tier. That is, proceed from the top downward within each work phase.

8. When deemed necessary (by project management), segment program development among several programmers. Such segmentation shall only occur at striped-module interfaces, and each programmer must be able to view the requirements placed by previous program levels on that striped

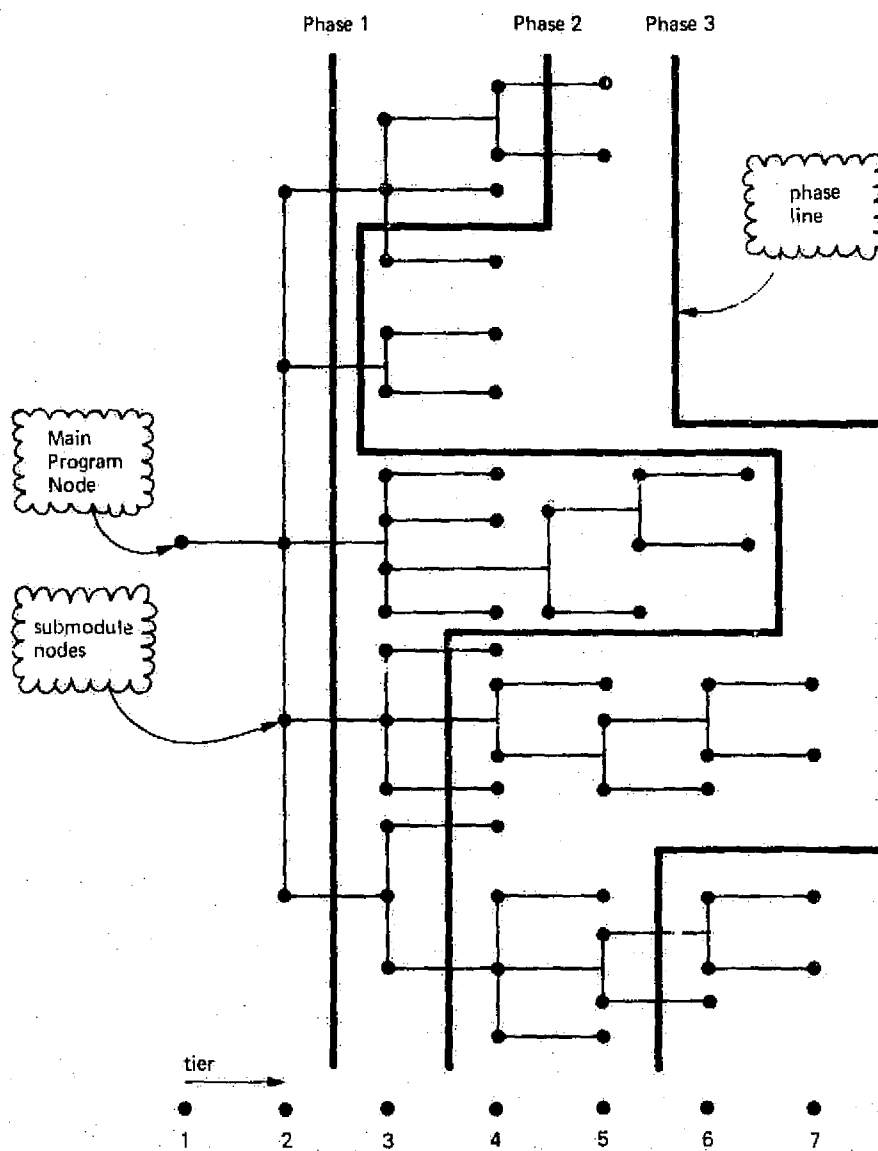


Figure 12-8. Formal design phases conceptually illustrated on the program tier graph as partitions of the graph that encompass more and more modules of the program

module as a total interface specification. If timing—either epoch or duration—is a commodity visible to other modules, it is to be included as part of the interface definition.

12.4 RULES FOR APPLYING STRUCTURED PROGRAMMING THEORY

Structured programming is not programming to eliminate “go-to’s,” although, in some languages, structured programs are “go-to”-free. Such is the case, for example, using CRISP. Rather, structured programming is conceiving a program in such a way that the need to think about having to “go-to” hardly ever arises. It is programming with logical structures that encourage clear thinking.

If a language possessed only the minimal canonical set of structures (sequence, WHILEDO, IFTHENELSE), one can argue that there are certainly times when control branches are needed. If the coding language is intended to be assembly language, this argument is even more intense. To maintain the aims of structure, I have, therefore, expanded the list of recommended structures, in flowcharting, as well as in their corresponding CRISP forms. Both flowcharts and CRISP isolate the designer from “go-to” thinking.

This section contains guidelines for generating the structured control logic hierarchy of the program. The development of this hierarchy takes place in-step with the development of the data structure and resource access hierarchies described in Section 12.2 (see Rule 5 of Section 12.1).

1. Arrange functional blocks naturally in a hierarchical design using only the control structures equivalent to those described in Appendix G. Each block has one entry point and one normal exit.
2. Use recursive subroutines (i.e., routines that call themselves or each other) only when a comprehensive statement can be made to assure that the required functional behavior is achieved. Every horizontally striped module in a structured design must be formally replaceable by its flowchart or coding.
3. In cases where alternative control structures perform identical functions, choose that which *best satisfies* the given ordered set of competing characteristics.

For example, when execution time outranks memory allocation considerations, one naturally minimizes memory

requirements and execution time whenever possible. One considers structures that reduce memory requirements but increase execution time only when (a) the loss in speed is a negligible fraction of the execution time of that module and the reduced core is a significant portion of storage for that module, (b) the module operates in an infrequently called, low-speed program branch in which the increased time is a negligible fraction of the branch execution time, or (c) the extra branching contributes to modularity and readability at a modest loss in speed or storage.

4. Whenever possible, design the program control logic and functions to favor high-speed, normal operation; if additional functions are required in order to achieve a structured design, place them in the slower-speed, less frequently used branch.

In some cases, the requirement for structure may be relaxed by the project management on a case-by-case basis, as, for example, in high-speed loops where the time to set, test, and reset structure flags would be a significant fraction of the loop timing. In such cases, however, the benefit and correctness must be clear.

5. Use a structure flag to record the outcome of a decision:

- a. When the decision result must be available later in the program but the data-space upon which the decision was based will be changed by then, so that the subsequent decision cannot be made correctly, or
- b. When the decision must be available later in the program and the flag-set and flag-test storage and/or time is less than that required to test the condition again.

6. When a structure flag is to be used in a loop to record the outcome of the loop-termination condition for later use, in compliance with Rule 5, above, preset the flag to indicate "unfinished" prior to loop entry, and then set it to indicate "finished" under the appropriate condition.

The flag test may be located either at the beginning or at the end of the loop, as illustrated in Figure 12-9. Other things being equal, placement of the flag test at the loop beginning increases top-down readability.

7. Consider using a stack structure to save loop-structure flags, as illustrated in Figure 12-10, below, unless the stack-unstack operation is a non-negligible fraction of the module execution time.

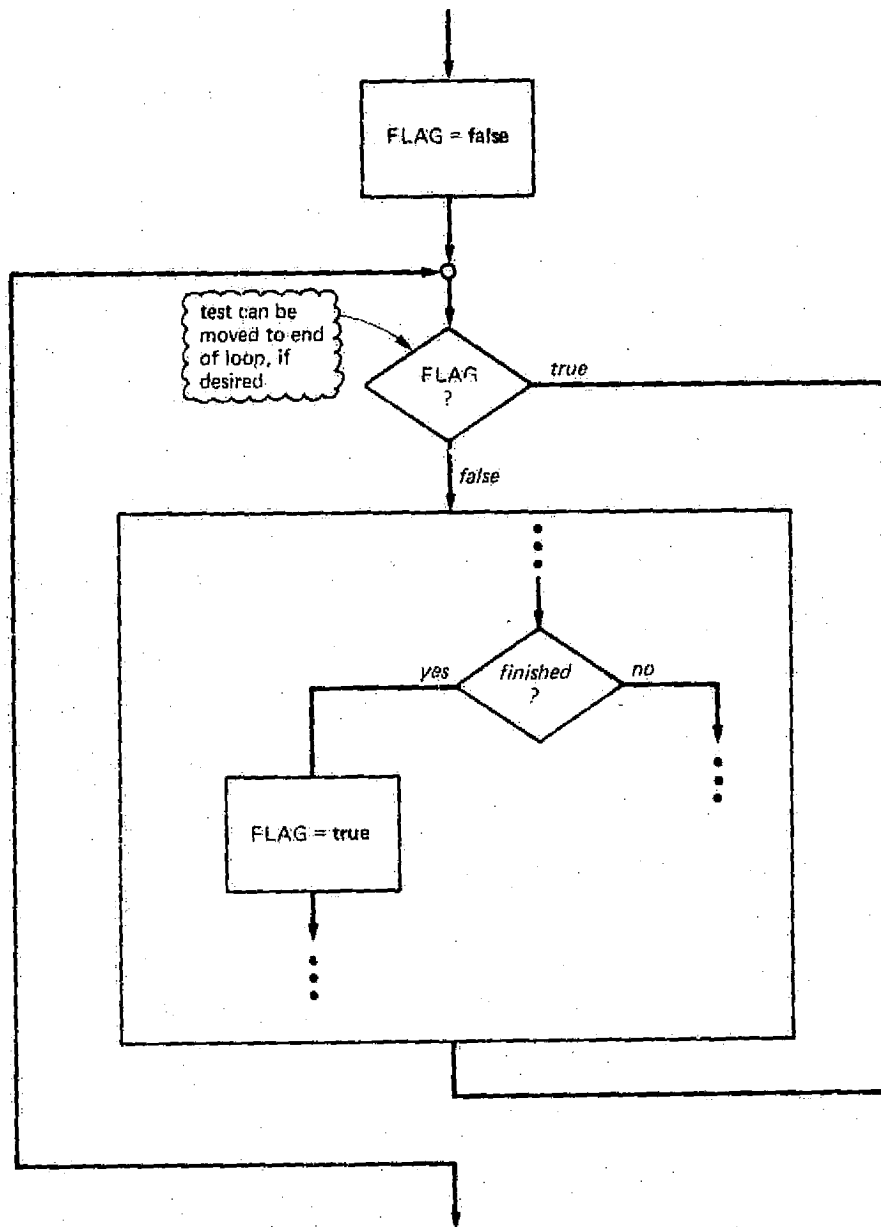


Figure 12-9. Loop configuration illustrating the use of structure flags to control iteration.

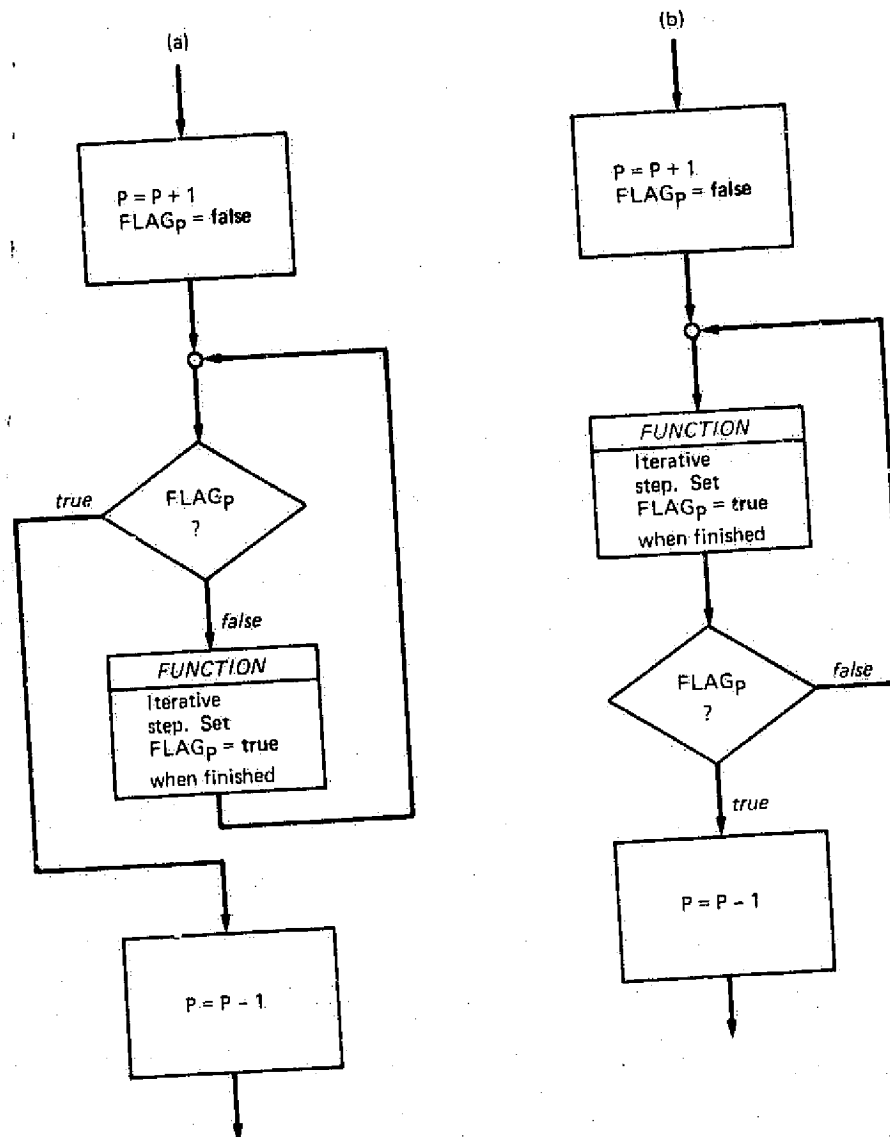


Figure 12-10. Use of a loop structure flag stack

This procedure is not only a convenience, since one need not keep track of all the loop flag names, but also somewhat of a savings in memory locations because separate flags are not needed for each loop. One loop flag stack suffices for the entire program. The value of the stack pointer is the level of loop nesting at any point in the program.

8. Do not use structure flags to force abnormal terminations to the same point as normal terminations. Abnormal terminations are defined as those in response to fatal errors which require that control be diverted to a program recovery mode, such as return to the user for subsequent decision making and manual operations. In cases where a recovery procedure partially or totally lies within the program, take steps to assure that the recovery procedure accommodates every state of the program from which the abnormal transfer occurs.

9. Avoid duplicate-code function blocks wherever possible, using subroutines or structure flags, as needed, except when:

- a. The function can be coded in fewer statements than needed for the setting and testing of structure flags, or than required in the subroutine calling sequence.
- b. The structure flag test time or subroutine linkage time is a non-negligible part of the module execution time.

For example, the two flowcharts shown in Figure 12-11 have the same program function; flowchart A has the function f shown twice (duplicated code), whereas flowchart B has removed the duplicate code by introduction of a structure flag.

10. Limit paranormal exits from canonic structures to situations that are either pathological, abortive, absolutely necessary, or outside the normal function of the structure.

12.5 RULES FOR REAL-TIME STRUCTURED PROGRAMS

The program structures for non-real-time programming form the basis for the separate, modular sequential segments of real-time programs; however, the added element of time interaction makes real-time program design a more error-prone activity, and even testing is inherently a more difficult and lengthy process. The rules that follow in this section are guidelines which reduce the mental capacity needed to comprehend and

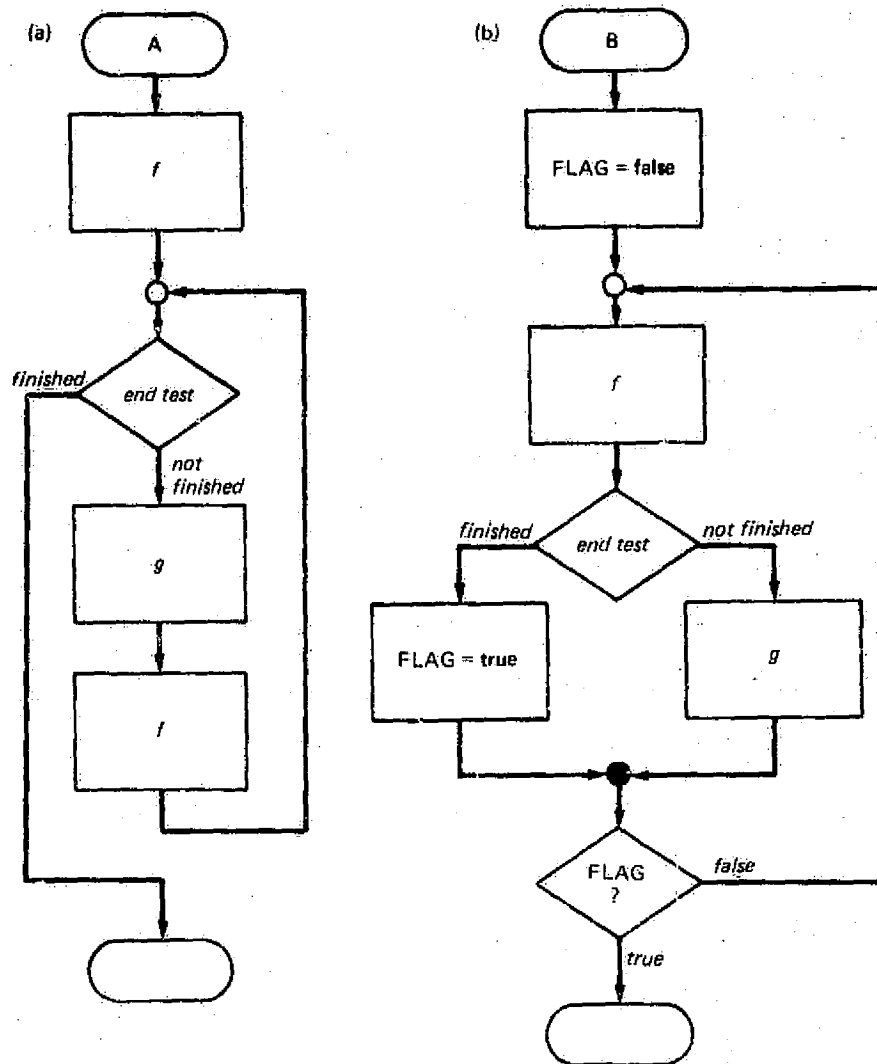


Figure 12-11. Two realizations of a looping function: (a) using a duplicated function f , (b) using a structure flag and no duplication

design such programs. They also promote original correctness and make verification testing a more practical and certain activity.

1. Avoid the use of interacting concurrent processes as a means of solving a programming problem unless it is necessary, as fixed by environmental requirements, or else unless there is a clear, identifiable advantage which outweighs the extra costs that will be incurred to achieve reliability.

2. Design concurrent or real-time, interrupt-driven modules using only hierarchic nestings of the structured forms given in Appendix G. Concurrent processes eventually merge upon normal termination of each, and proceed only when all component subprocesses have terminated normally.

3. Identify or define the level of access in the system nucleus that will permit processes to communicate, synchronize, and allocate shared resources on a mutually exclusive basis. If these are not provided by the supplied operating system, devise methods to implement these either into the operating system or into the program being developed.

4. Orient program designs toward developing consistent programs only (see Section 6.2). Assess consistency as well as program correctness during the design process.

5. Strive to design processes in terms of functions that can be easily tested on a practical real-time basis.

6. Define synchronization and communication requirements for processes consistent with Section 6.2.4.

7. State a policy or design rule for each set of shared resources that will make the corresponding processes deadlock-free. Prove that each such policy is sufficient to prevent deadlocks.

8. Set interrupt priorities for hard-real-time processes to assure that process deadlines are not missed. Use look-ahead feasibility studies to estimate the likelihood of timing errors in critical-deadline modules, but maintain the top-down formal development discipline.

9. Arrange, where possible, to monitor deadline status and to flag timing errors. Strive to design the program so that the program degrades gracefully, but does not fail, when deadlines are missed.

Design for the worst case. In some cases, the allocation of more hardware (e.g., CPUs) to a particular set of concurrent tasks is possible, and is often more cost effective than trying to fit too many things into a single restricted set of hardware.

10. Identify to what extent resource protection (Section 6.2.3) is needed in the program being developed and to what extent the operating system fulfills these needs. Accommodate, to the greatest extent possible, those needs not provided by the operating system by setting appropriate programming standards.

11. Do not attempt to gain arbitration of interrupt processes by reassignment of priorities (or interrupt disabling) except in cases where time is at a premium and there is a reasonable assurance that the program integrity (or consistency) will not be violated. Similarly, do not relax the requirement for consistency unless it can be shown that any inconsistency is momentary, and performance lies within required limits.

12. Identify interacting resources and synchronization requirements among mutual resources early in the top-down development of the program design, preferably at the same level that displays the program fork (Figure 12-12).

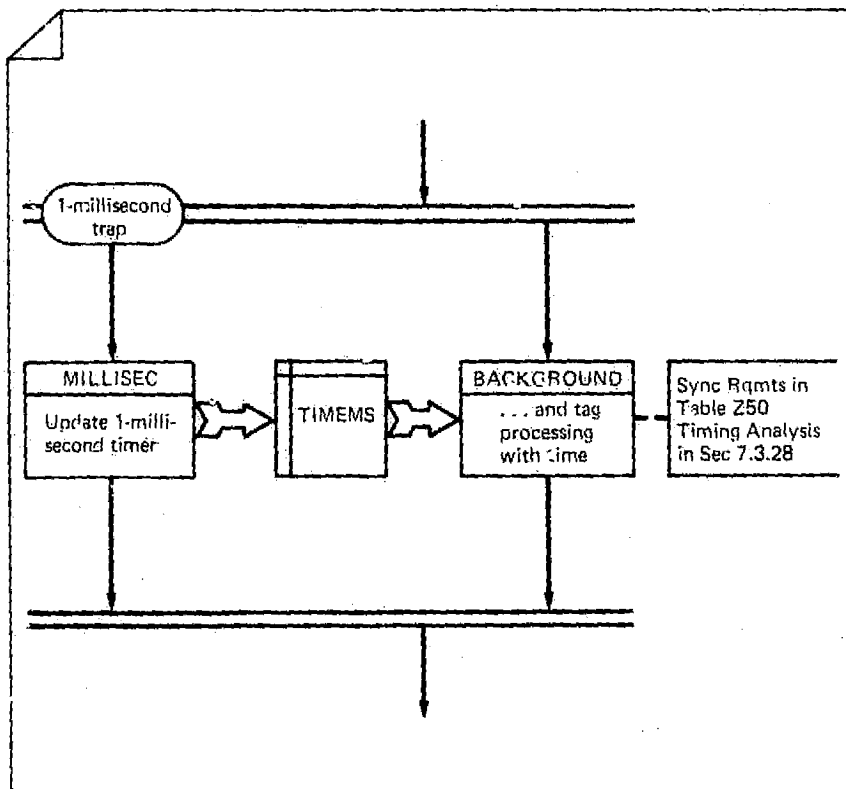


Figure 12-12. Identification and documentation of interacting resources and synchronization requirements at first level in top-down development where processes fork into concurrent activities

12.6 STANDARD DESIGN PRACTICES

This section contains a few general rules for developing specifications that relate to the program quality, as reflected in its reliability, maintainability, generality, usability, modifiability, and performance.

1. Use standard, already proved (and documented), structured algorithms and code whenever possible.

2. Minimize the functions in each branch following a decision node by locating above the decision node all functions which logically precede or operate on data sets that are independent of the branching condition. The first function in each branch is thus dependent on the branching having been executed. Locate below the corresponding collecting node all subsequent functions that are independent of the branching.

3. Do not use identical (or equivalent) test conditions in succeeding branching nodes if the data on which the decision is made has not changed; this is an indication of unnatural, artificial structure.

4. Provide programming standards to prevent the inadvertent misuse of common resources, such as variable, label, and file names, storage occupancy, I/O and interrupt facilities, etc., when not provided by the programming system.

5. Design programs or subprograms defined by decision tables to provide complete testing of the ELSE-rule, unless (1) an ELSE-event is impossible, or (2) the consequences of not testing the ELSE-event are identifiable and justifiable on a case-by-case basis.

6. Specify or build flexibility into the program to allow changes in tolerances, formats, and to adjust criteria for control decisions without changing the basic structure of the program.

Designing for flexibility does not mean designing for generality; generality tends to weaken module cohesiveness, whereas flexibility does not. However, when a general approach is as easy to implement as a specific one and exhibits the same cohesive class, design for the general case.

7. Provide options in running the program to accommodate fallback methods for severe conditions. However, avoid options that may overly complicate or seriously degrade performance. Specify a no-stop, no-abort mode for programs processing data, so that the program need not stop

because data quality has degenerated. If a stop capability is programmed, make provisions to retain the data and work up to that point. Provide a means for continuing the program at the next step, unless to do so is not meaningful.

8. Design the program to have default values for control options so that the program can be run in its normal mode without the need for control inputs.

9. Check for errors in data passed between major modules (especially if separately compiled or designed by different individuals). Include self-checking and monitoring features within critical modules.

10. Strive to design the control logic of a module so as to be independent of the way data is stored, and modularize accesses to data structures so that, if data is restructured at a later time (e.g., for more efficiency), only the access modules need to be altered.

12.7 RULES FOR DOCUMENTING STRUCTURED SPECIFICATIONS

Documentation of the procedural and data design becomes the programming specification, and, once this has been coded, checked, and delivered, it becomes the "as built" specification. The design process thus culminates in a body of information contained in the Programming Specification portion of the SSD.

The extent to which the SSD gets filled in and the order in which the filling gets done is set by project management in phases as described in Section 12.3 and Chapter 10. Having a comprehensive SSD outline, such as that in Appendix E, even if it is not going to be filled in completely, can be a tremendous boon in the design, because the designer can use it as a mental jog, filling in each item in his own mind, deciding how things should be done. If he cannot respond to each item, even in his own mind, then he is probably going to have trouble somewhere along the way. A graphical sketch of an SSD with emphasis on the Programming Specification appears in Figure 12-13 (Figure 11-5 contains a graphical outline of the SFS portion of the SSD).

I will assume here that a complete program internal specification consists of a combination of flowcharts (or equivalent), narrative descriptions, data-structure definition tables, correctness assessments, etc., and I give documentation rules for each. The level of detail described here will later be labeled as "Class A" documentation in Chapter 16. The actual amount

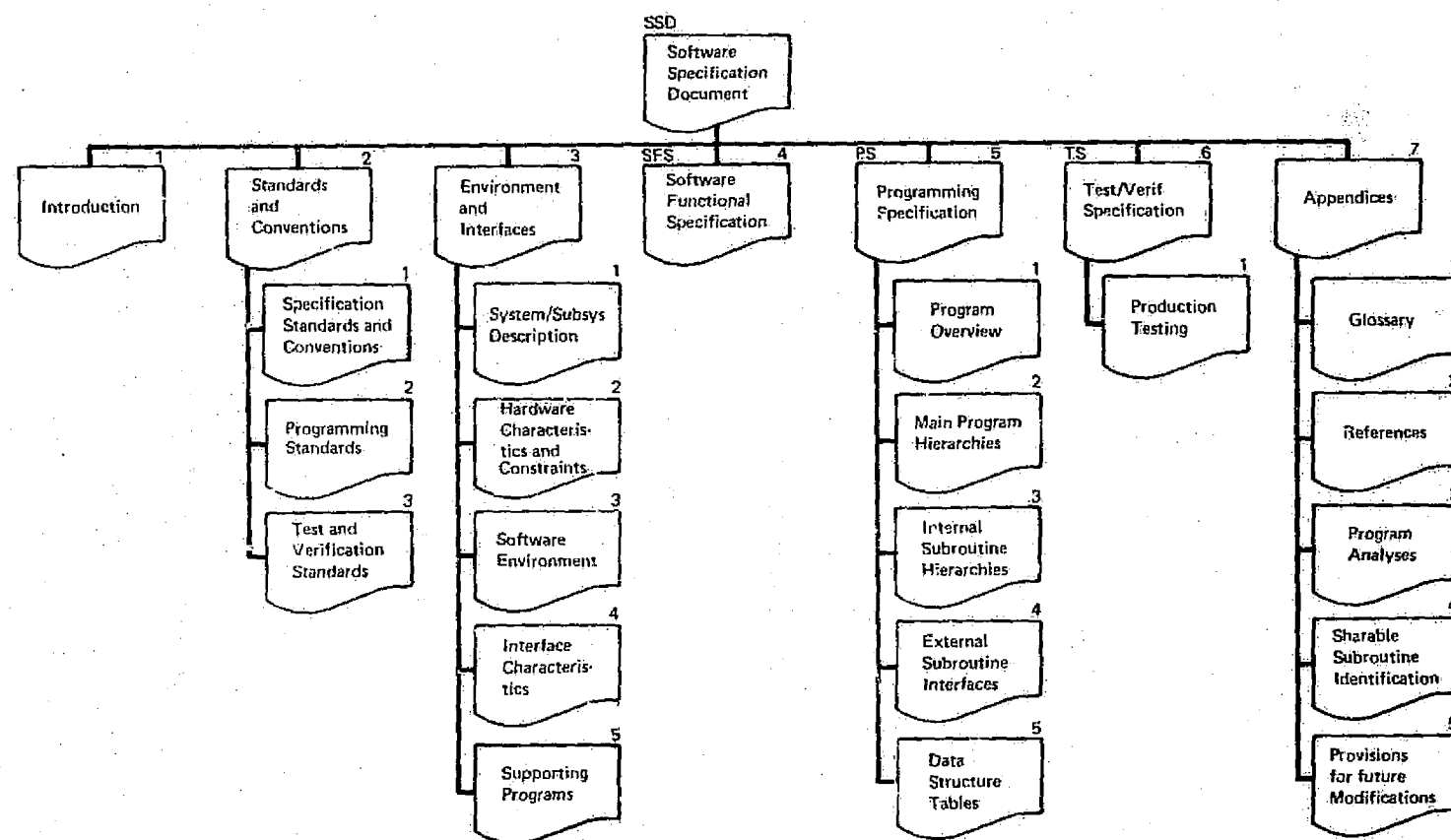


Figure 12-13. Graphical outline of the SSD, with emphasis on the program internal design

and type of documentation for any given project, however, may be more or less than what I specify in the remainder of this section. Guidelines for defining a full range of levels of documentation appear in Chapter 16.

12.7.1 General Rules for Documentation

In the guidelines that follow, the term "module" refers to a flowchart (or equivalent) and its accompanying narrative. The flowchart displays the control logic and operations making up the algorithm and the narrative extends, explains, and expands upon the procedural material. When feasible, these may be arranged as illustrated in Figure 12-14 for readability.

The overall guideline that has governed the development of these documentation rules is the following (for "Class A" detail):

Documentation of each module should exist to a sufficient degree that correctness can be assessed for each individual module, formally on the basis of its control-logic, and by audit for its functional completeness. Specification of data structure detail and formats internal to a module, as well as requirements relative to accumulated numerical and timing errors, may be deferred to a later level of the documentation, if appropriate.

This section also contains guidelines to hasten the documentation process.

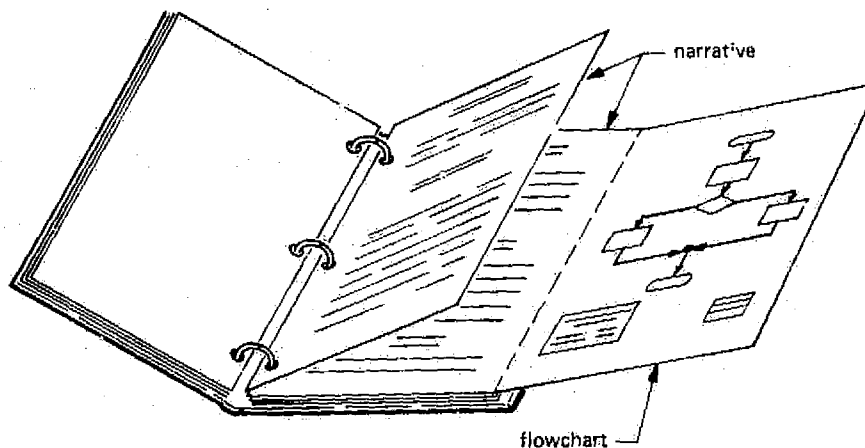


Figure 12-14. Insertion of flowchart in the SSD as a foldout on the last narrative page describing the flowchart can enhance readability when only one side of the page is used.

1. Make design documentation form the interface between design and coding activities; however, do not get in series with the clerical activity of doing the formal documentation.

It is permissible to code from approved hand-written specifications while the documentation is being typed or drafted, for example (see the method illustrated in Figure 10-7).

2. If working documentation can be entered and maintained in computer files, rather than produced by hand, then by all means do so, in order to minimize documentation lag and to promote project design visibility.

3. Strive for clarity. Remember that documentation is for communicating ideas to people. Therefore, specifications should discuss the program in terms common to the developer, coder, later maintenance personnel, and, perhaps, the user as well. Even formal, unambiguous language specifications may very often require natural language explanations, graphical aids, and examples for understanding.

4. Strive for completeness without redundancy. Reading a design should not require a translator, nor learning a large data base, nor consultation of the listings.

5. Specify *control-logic* for a given module completely, so that module control can be assessed for correctness, or coded and tested, with no other aid than references to preceding levels of the procedural design and to material in the non-procedural sections of the Software Specification Document at the current development phase.

- a. All decisions are to be explicit and determinable within each individual module; there should be no need to refer to deeper levels of design or code (see Figure 12-15).
- b. Unstriped modules must have explicit values, conditions, and reasons stated for all control flag assignments.
- c. For striped submodules that alter one or more control flags for the current module or a parent module, state the explicit values, conditions, and reasons for all control flags altered.

6. Document the *functional characteristics* of a module to that point which permits an audit of the algorithm of the current module against its stated function and an assessment of functional correctness. Specifically,

- a. Unstriped modules are to be described explicitly enough to permit coding without functional ambiguity and without reference to deeper levels of design or previously completed code.

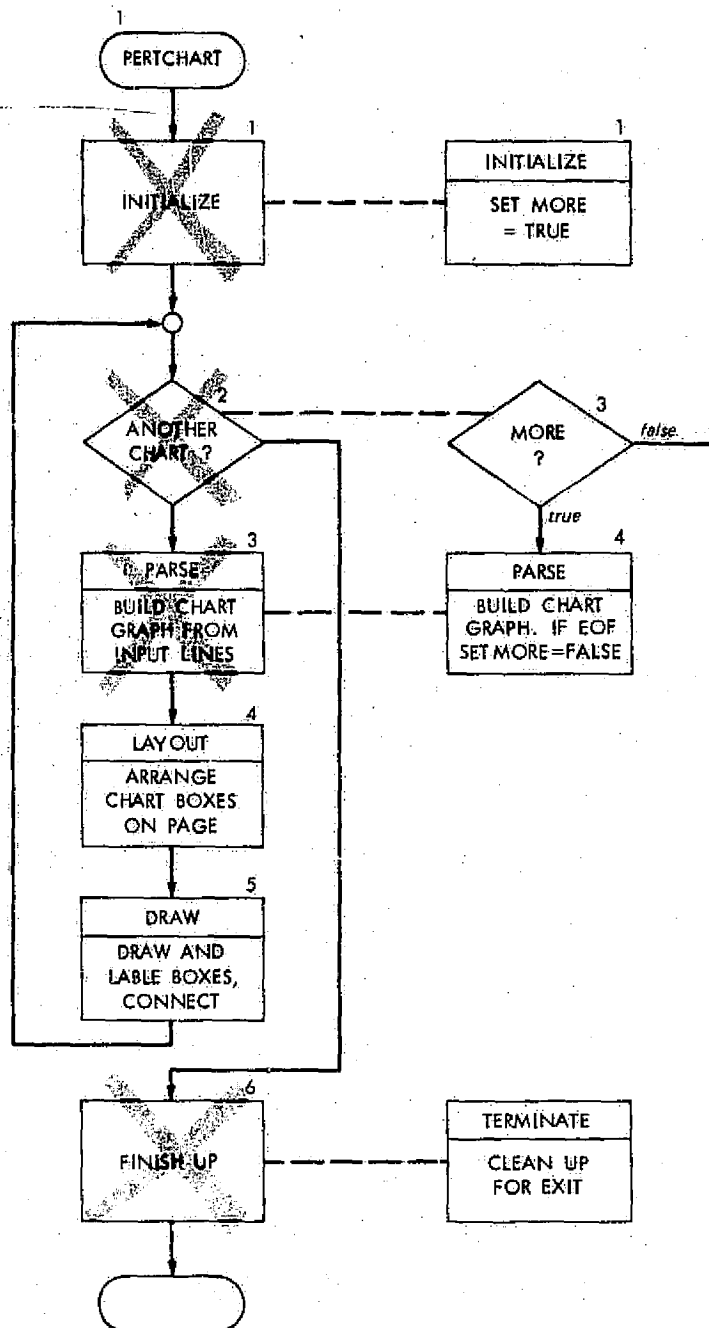


Figure 12-15. Making a module complete enough for control logic proof of correctness and immediate codability (crossed-out boxes are to be replaced by those connected by dashed lines)

- b. Interfaces of external subprograms that are not part of the current design (vertically striped modules) are to be described explicitly, in sufficient detail that any subprogram satisfying the stated interface characteristics will operate correctly in the current design. References to documents that provide additional information may be included but explicit interface characteristics must always be described in the SSD.
- c. Striped symbols of the current module that receive control flags from the current and/or parent modules must have accompanying narrative explaining all conditions, values and actions for each setting of the flags.
- d. Functional descriptions of striped symbols may be quite broad, but must be complete, and tell what actions that submodule performs. No function may appear in a module at level $n + 1$ that is not a component function of that module as described at level n .
- e. Module functions should be described using assertions defining the computer state upon entry, the specific action that takes place, and the relevant state of the computer on exit, including time or sequence dependencies, quantitative measures, and interrelations among interfaces.

7. Declare *data structures* accessed by unstriped submodules as to specific type attributes necessary for coding in the intended programming language at the current flowchart level without ambiguity. Definitions of internal data structures accessed by striped modules, not pertinent to control logic or functional correctness, as specified in Rules 5 and 6, above, may be refined in later levels in the design. These successively provide more and more detail about the data structures and requirements involved. Specifically,

- a. Use the most descriptive names for data structures permissible within the programming language, especially when such names eliminate the need for explanatory narrative.
- b. Make each further detailing of a data structure definition consistent with every previous assumption concerning its use. As a minimum, the final, explicit form of a data structure definition must contain: (i) the structure name; (ii) its mnemonic derivation; (iii) type attributes (e.g., real, string/array variable, simple variable, etc.); (iv) range of values; (v) scope of activity (i.e., over what portions of the program the structure is not available for reassignment or reuse by other parts of the program); (vi) description of the use of the data structure in the program; and (vii) a list of any data structures that share or overlay storage with this structure.

- c. Declaration and/or initialization of a new data structure may appear as an entry requirement of the current module, to be performed in specified, previously defined modules. Document such actions in the narrative or by annotations to the flowchart for the current module. Locate the actual declaration/initialization code within the specified modules (striped or unstriped), indented (if permitted by the programming language) to show that it is a later addition to that module (not contributing to nor detracting from the previous assessment of correctness) and annotated to indicate the later module which requires this initialization.
- d. Data structures may be referred to in striped modules in generic terms when not related to control-logic correctness. Assumptions made in such references must be consistent with the current state of the data structure definition.

For example, a striped module may state that a set of characters is "put" in the "name table," whereas an unstriped submodule which implements that function must be specific, as "NPTR=NPTR+1, NAME(NPTR)=NS," in which NPTR, NAME, and NS appear as appropriately detailed declarations in data structure definitions.

- e. Describe the form and content of data structures, and define relationships among data items.

For example, one such relationship might read "field 2 has meaning only if field 1 is non-zero."

8. Identify I/O requirements that interface with internal data structures or other I/O requirements. These must be specific for unstriped modules, but may be referenced in appropriate generic terms for striped modules. Specifically,

- a. An I/O interface is to be defined so as to be consistent with every assumption concerning its use.
- b. Any I/O interface not made specific at the current level should be maintained in an appropriate I/O Requirements Table in the Software Development Library or Project Notebook. A complete description of the I/O interface must eventually be included in the SSD (see Section 12.7.4, below).

9. Identify all constraints appropriate in the implementation of the module, such as:

- a. Critical maximum time of execution.
- b. Data ordering.
- c. Machine timing characteristics.

- d. Special hardware features.
- e. Accuracy requirement as it relates to computer word size and the need for single- or multiple-precision computation.

12.7.2 Rules for Documenting Structured Flowcharts

As I indicated in Section 7.4, generating flowcharts and maintaining these by hand in an evolving program design can be time-consuming, non-productive tasks. Yet, many feel that there is great benefit in having such charts for their graphic value. In Chapter 17, I discuss automatic production and maintenance of design flowcharts from CRISP-like source statements, as a natural, easy, flexible and productive way of generating high-quality charts.

The rules given here describe the format, content, cross-referencing technique, etc., of such charts. The detail level corresponds to "Class A" standards (Chapter 16), which may be relaxed during the architectural phase, at project option.

1. Limit flowcharts to one 21-1/2 × 28 cm (8-1/2 × 11 in.) page each, except in unusual circumstances, such as when a single decision results in multiple branches that will not fit on one page, and when the symbol-stripping convention only adds to confusion.
2. Conform flowcharts to ANSI Standard X3.5-1970 augmented as given in Appendix B; symbols should not be varied in proportion. Symbol text should be brief, but exact. Use narrative accompaniment to extend, explain, and expand upon the symbol function.
3. Give each chart a Dewey-decimalized number that identifies its location in the design hierarchy. All symbols on the chart are numbered consecutively. A possible exception is the numbering of collecting nodes. The numbering system recommended is the pre-order traverse (Section 5.1.3.3), or top-to-bottom, left-to-right order. The symbol number is placed at the upper right of the symbol.
4. Use cross-reference identifiers located at the upper left of flowchart symbols corresponding to invocations of subroutines and major program segments. Subroutines should use an alphanumeric level-1 identifier. Major program segments may be given an integer level-1 identifier.
5. Uniquely identify symbols by concatenating their chart number to the symbol number (on the upper right). Thus, the symbol labeled 5 appearing on chart P4 would be designated P4.5.

6. Enter the name of a striped module so that it appears above the stripe; it appears again inside the entry "bubble" of the flowchart at the next level.

Figure 12-16 shows a set of hierarchically nested subprogram flowcharts, annotated in the correct manner. In this example, 1.2.5.1.6 refers to a striped box labeled 6 on chart 1.2.5.1 at level 4 (since four level numbers make up the chart number). The entire chart 1.2.5.1 is an expansion of the striped box numbered 1 appearing on chart 1.2.5 at level 3, which in turn is an expansion of striped box number 5 of chart 1.2 at level 2, which is an expansion of box number 2 of chart 1 at level 1.

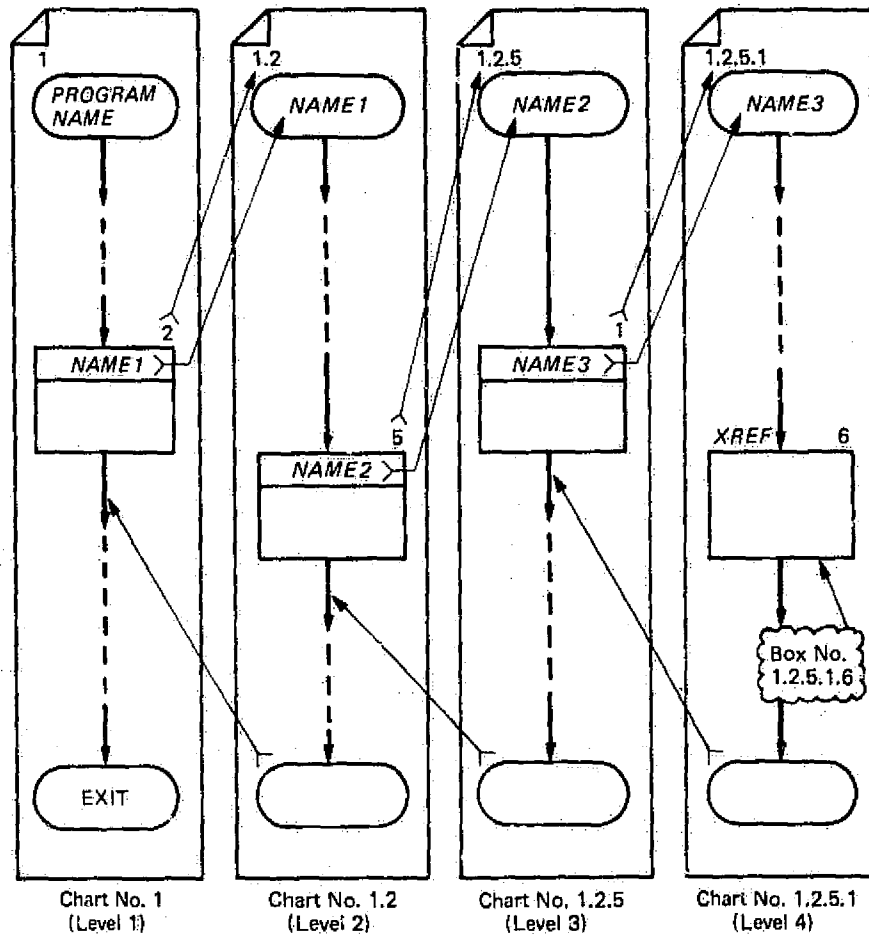


Figure 12-16. Hierarchically nested subprogram flowcharts

An unstriped box, such as 1.2.5.1.6 in Figure 12-16, indicates that further detail is available only in the accompanying narrative or at the code level, and the cross-reference, *XREF*, if supplied, gives a reference to a section in a programming manual or systems manual.

Striping conventions are summarized in Figure 12-17.

7. Distinguish *subprogram* flowcharts by chart exits having a blank normal exit "bubble," or by chart exits labeled "EXIT" or "ABORT," indicating paranormal or abnormal termination.

8. Use unstriped process symbols to state precise, functionally unambiguous, and explicit specifications for coding, as space permits. Use narrative keyed to the symbol to explain or expand upon the material given within the symbol, if needed for understanding. Strive to make the symbol contain enough information for coding without recourse to the narrative, to increase coding speed.

9. Distinguish *subroutine* (and *function*) flowcharts by chart exits having "RETURN" or "ABORT" in the normal exit symbol. Each callable function then has its own hierarchical structure, as specified in Rule 2, above.

Cross-referencing conventions are summarized in Figure 12-18. Note that a cross-reference code, when it appears, takes precedence over the box reference code.

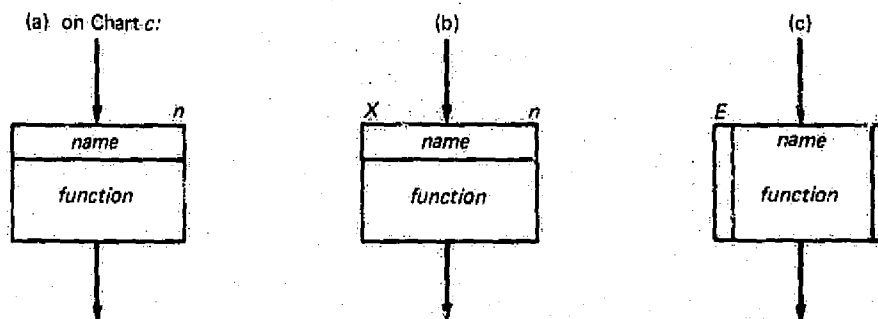


Figure 12-17. Module striping conventions: (a) internal subprogram or procedure whose expansion appears as chart *c.n* elsewhere in this set of documentation; (b) internal subroutine or major subprogram whose expansion appears beginning on Chart *X* elsewhere in this documentation; and (c) external subprogram or function whose expansion is not flowcharted in this set of documentation, but whose precise interface specifications are given in section *E* of this set of documentation

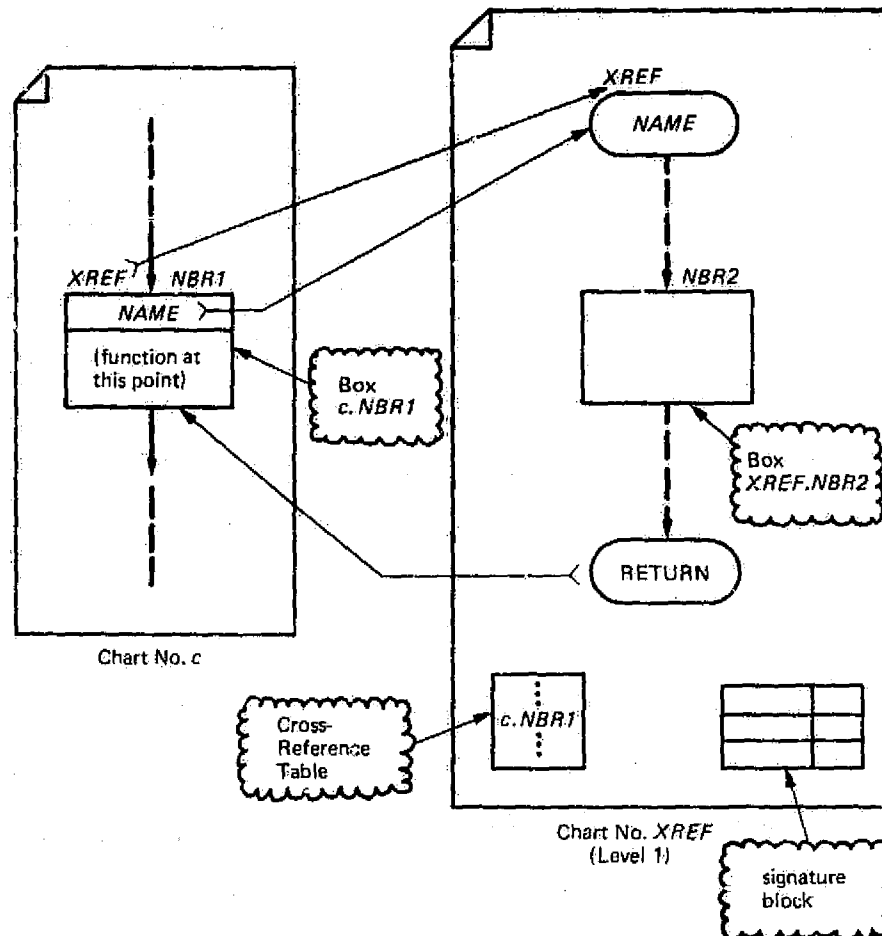


Figure 12-18. Subroutine and major subprogram annotation method

10. Attach or append a table to each top-level subroutine flowchart and each top-level flowchart of a major program segment, giving a list of the cross-references to invocation points within the program, as illustrated in Figure 12-18.

This table, filled in as the design progresses, serves to ensure, in the event that later changes are made in the functioning of a subroutine, that the side effects can be identified.

11. Cross-reference the use of a function within a symbol by either placing the appropriate identifier at the upper left of the symbol, or by attaching a separate striped symbol to the invoking symbol. Figure 12-19 illustrates these cross-referencing methods. Cross-references to standard library functions are unnecessary.

12. Annotate the entry line of a flowchart with any entry requirements to be placed on a module at an earlier documentation level.

Such entry requirements are normally initializations of variables which must be made so that the algorithm of the current module is operable. Since the initialization pertains only to the execution of this module, and not to the modules at previous levels, the annotation lends readability to the design, placing the relevant information in the relevant place. Figure 12-20 shows an example. Additional material usually appears in the accompanying narrative.

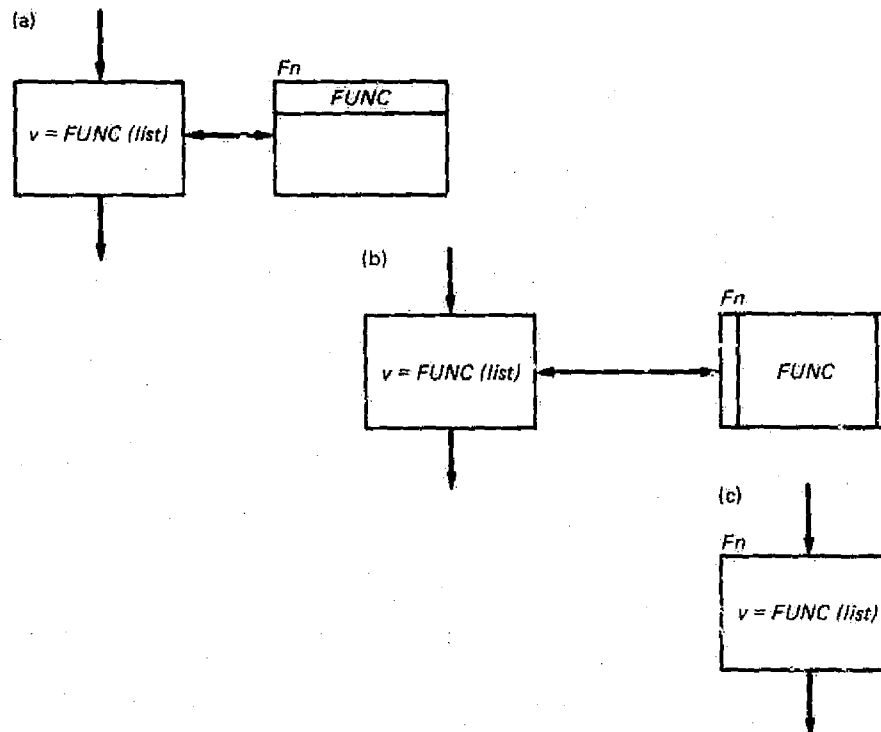


Figure 12-19. Flowchart convention for computations involving an (a) internally defined, (b) externally defined, or (c) standard library function *FUNC* with argument(s) *list* (designator *Fn* is the cross-reference chart number of interface description)

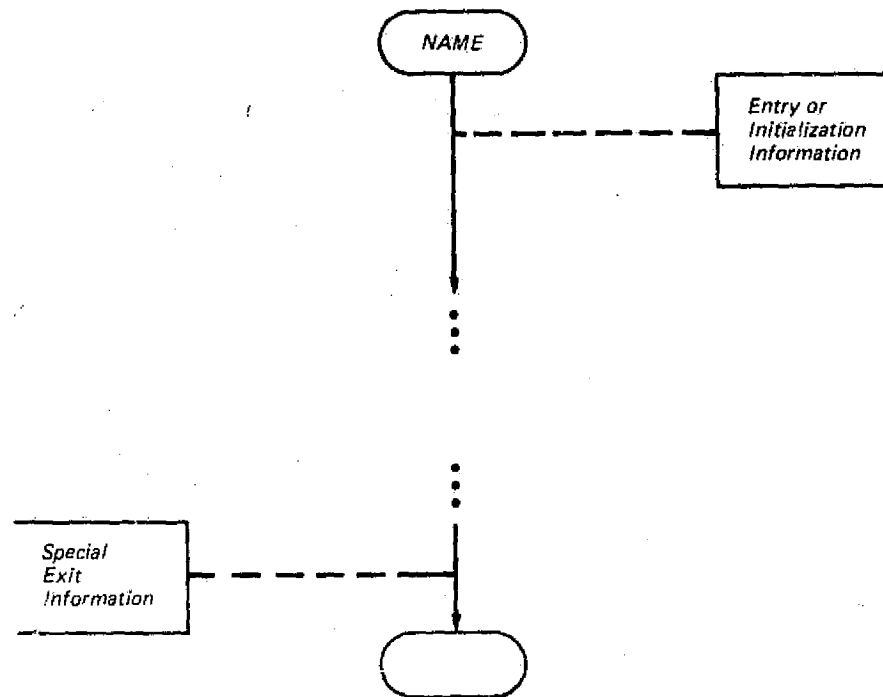


Figure 12-20. Annotation of flowchart to show conditions for entry, initialization information, and exit information

13. Indicate for each decision box the explicit condition to be tested; give the answers (as well as meaningful explanations when it lends to program clarity) on lines from the decision box. Decision boxes may not be striped (see Figure 12-14).

14. Draw flowchart branches to reflect left-to-right logical consistency: for binary decisions, draw *true* (Yes) to the left and *false* (No) to the right; for multiple branches, arrange decisions in a logical CASE-order, as shown in Figure 12-21.

15. Locate the collecting node corresponding to a decision branch directly under the decision box vertex, as indicated in Figure 12-22, below, even if one of the branches leads to an abnormal or paranormal termination rather than back to the collecting node.

16. Locate the flowline exiting from a loop directly below the loop-entry flowline, as illustrated in Figure 12-23.

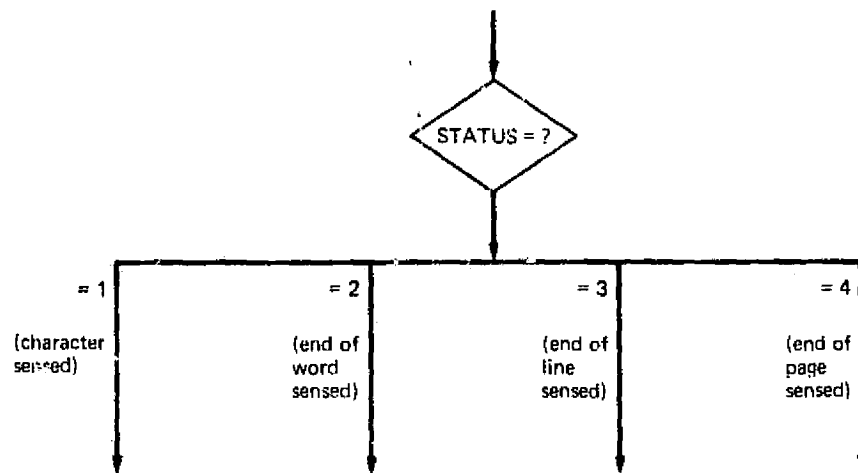


Figure 12-21. Use of explanatory comments on decision branches to clarify meaning of branching variable values

17. Place only one arrowhead on each flowline, at its termination.

18. Affix to every flowchart a configuration-control box, which will contain approval signatures and approval dates of the module designer, the design verifier, and the project manager or his designate.

12.7.3 Rules for Documenting Flowchart Narratives

The principal reason for having flowchart-accompanying narrative is that the flowchart boxes are generally too small to contain both complete specifications of the functions alluded to within them and descriptions of the significance of algorithmic steps. The rules that follow circumvent this lack by providing an orderly format for writing down what should be said in each box for understanding and assessment of program correctness.

1. Arrange the narrative at each level to have the following format, as illustrated by Figure 12-24 relative to the flowchart shown in Figure 12-25.

- a. Begin each module description with its name, chart number, and date of latest change as part of the design document section title. Identification information should be placed in the upper right corner of each page. The following configuration is suggested:

Chart Number (decimal number)
 Module Name
 Date (date prepared)
 Page__of__ (identifies the number of narrative and flowchart pages for the module)

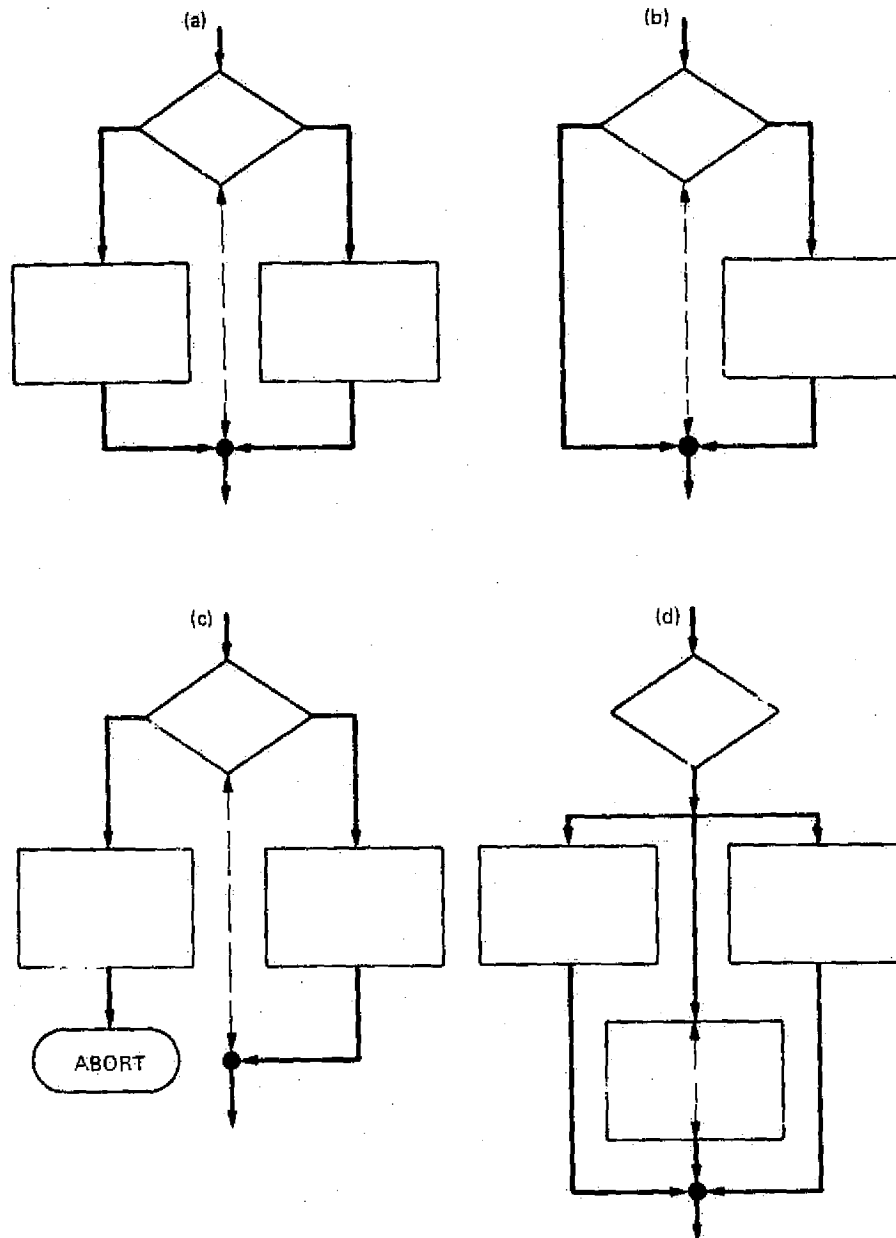


Figure 12-22. Consistent format for alignment of collecting nodes following decision nodes

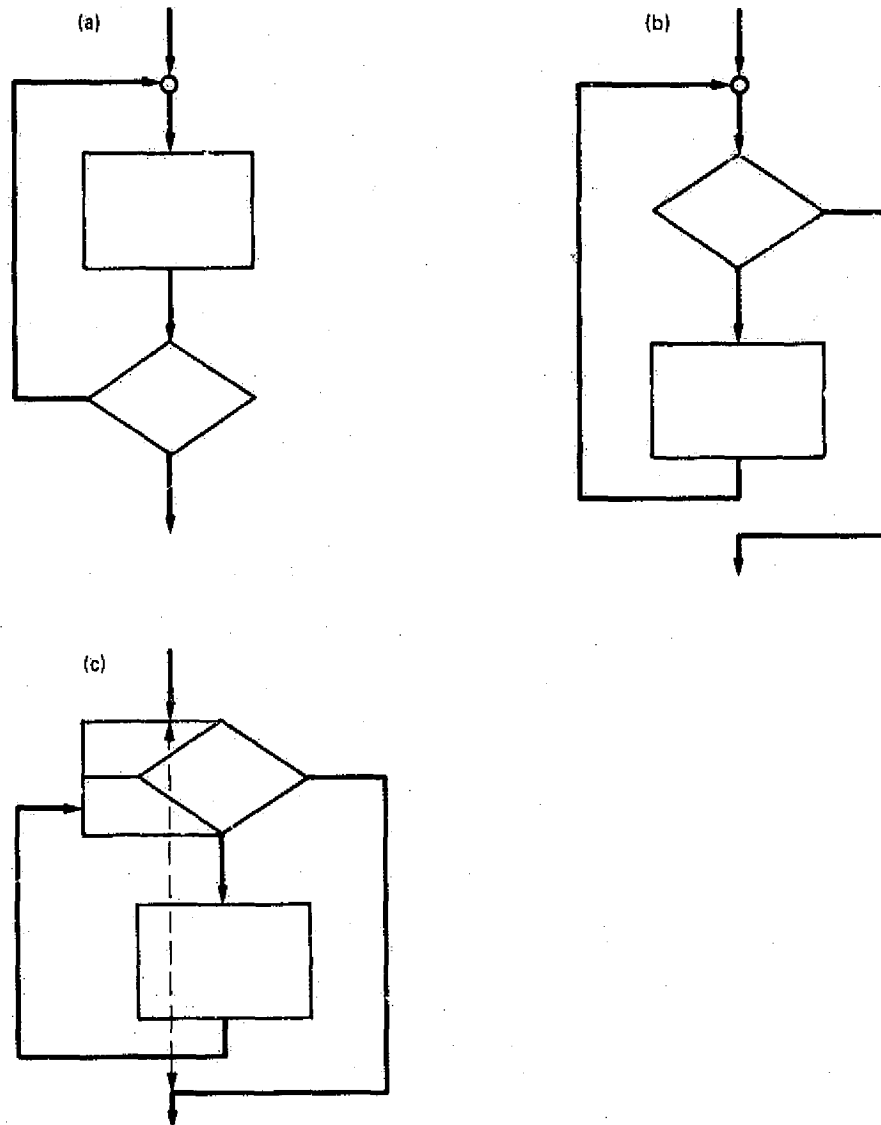


Figure 12-23. Consistent format for alignment of flowline for loop exits

SSD-DOI-5466-SP

Chart 4.4
FINDNO
8/7/75
Page 1 of 2

5.(4.4.1) FINDNO Procedure

On entry, an input line has been received into an input buffer via MSGIN (U14). The Line Pointer, LNPT, is positioned so as to extract the first character of the line.

This procedure discards leading blanks, if any, and looks for an MBASIC statement number. If a statement number is present, the digits are converted to an integer and placed in the Value variable V; the Class variable C is set to 11 to indicate that the first symbol on the line is an integer. If a statement number is not present, C is set to 0, indicating the input line does not have a statement number; V retains its entry value. See Table 7.2.2.5 for further symbol class and value definitions.

On exit, C and V contain the symbol Class and Value values above, the Current Character variable, CCHR, holds the character which stopped the number scan, and LNPT points to the next input character to be fetched. A value of V>999,999 indicates the statement number is too large.

- .1/U17 GET the first character from the current input line (see MSGIN/U14) into CCHR.
- .2/P4 CATEGorize CCHR. If blank, scan to and fetch the first non-blank. Set Character Class, CC, and Character Value, CV, as specified in Table 7.2.2.5. CC will be 2 if CCHR is a digit, and CV will contain its integer value.
- .3-.4 If the first non-blank character on a line is not a digit, set C to record that no statement number is present, and exit.
- .3-.5 If the first non-blank character is a digit, set C to record that a statement number is present. Initialize V to the first statement-number digit as an integer, and set the loop structure flag FIND SWtch, ENDSW, so as to iterate, bringing in digits.
- .6-.14 Bring in remaining digits and convert them into V as follows. After GETting each character:
 - .7-.9 If the character is blank, set ENDSW to terminate the scan, as the statement number has now been extracted, and exit.
 - .10/P4 Otherwise, CATEGorize the character seen, as in step 2 above.
 - .11-.12 If not a digit, terminate the iteration, and exit.
 - .13/E143 If CCHR is a digit, ACCUMulate its value into V. Set ENDSW=1 to terminate the iteration if V exceeds 999,999.
 - .14 Iteration continues until all digits have been processed or until overflow was detected.

S-39

Figure 12-24. Sample narrative format and content for a module FINDNO, shown in Figure 12-25

ORIGINAL PAGE 1
OF 1002 QUALITY

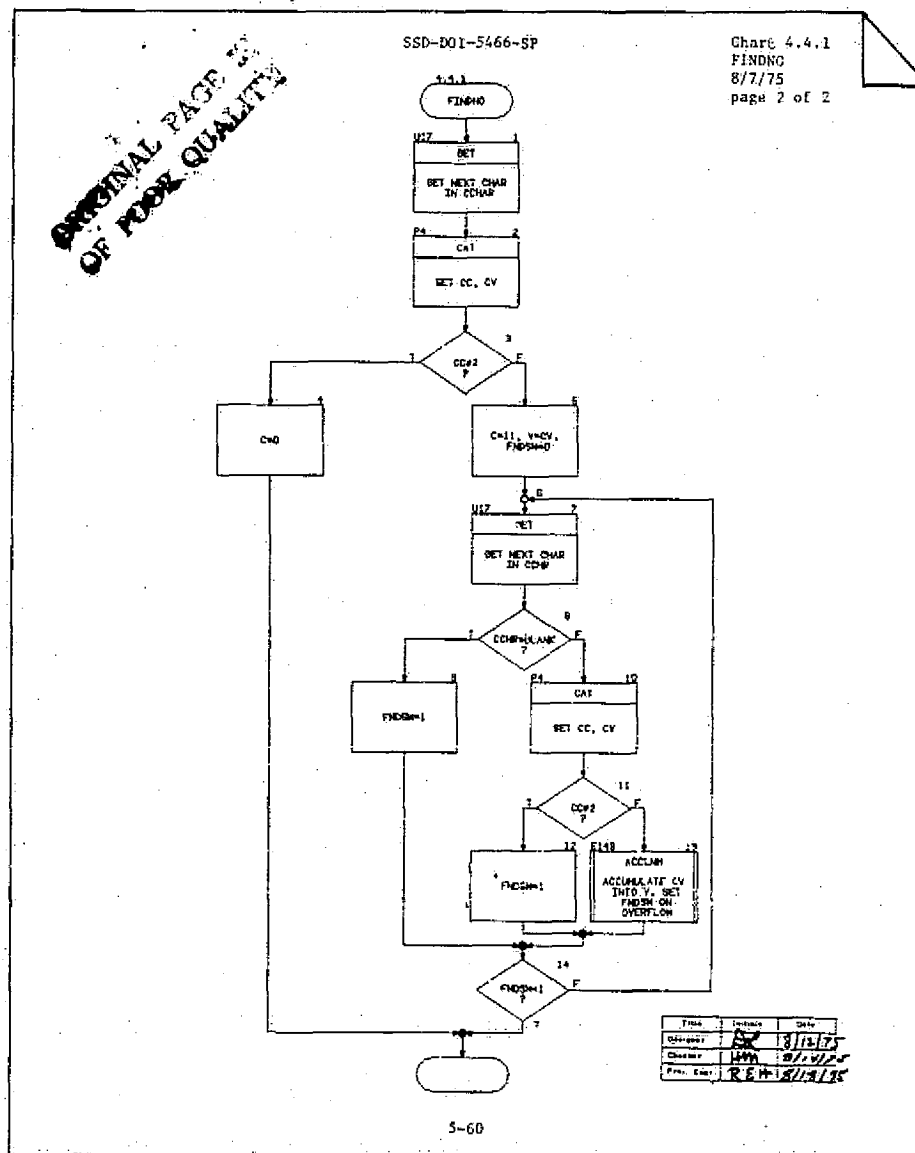


Figure 12-25. Standard flowchart corresponding to narrative in Figure 12-24

- b. Prior to the description of the algorithm of the current module, state all assumptions on inputs, common data, etc., needed for understanding this module. Describe the module function and its outputs, actions, and constraints.
- c. Key each of the algorithmic steps to the symbol numbers as they appear on the flowchart; these numbers appear in the left-hand margin of the narrative as *.n*. Subroutines or major subprograms invoked may be distinguished by attaching their cross-referencing level-1 chart number following the step number, as *.n/Sm*.

2. Do not include irrelevant or extraneous information. Be brief, and do not repeat information that exists on the flowchart. Use references rather than repeating information appearing in parent modules or in auxiliary tables. Functional detail given in a parent module may be repeated, if desired, however, when needed to:

- a. Identify which subfunctions belong to which symbols on the current flowchart.
- b. Detail a given function into subfunctions.
- c. Clarify the current execution state (for example, by giving an explanation of the meaning of a certain condition that has caused entry to the current module, etc.).

The narrative need not include a description of every numbered symbol on the chart, so long as the overall documentation is comprehensive. Descriptions of groups of symbols can be used when more meaningful.

3. In specifying actions taken within a module, use single-spaced text in the imperative mood; avoid using the passive voice except in occasional explanatory statements regarding interfaces, conditions, etc.

For example, the narrative description of a module "PARSE" within a BASIC compiler might read as follows:
"Accept statements and commands entered from the terminal with normal or line number prompting, or from a file without prompting during LOAD, MERGE, or COPY (to workspace). Parse statements and commands, and translate statements into an executable pseudo-op form. Terminate abnormally if errors are found, print an appropriate error message, and return to the command mode."

4. Use descriptive mnemonic names for all data structures. Provide the mnemonic derivation of all data-structure names used in a module at least once (at its first top-down appearance). Names appearing in tier chart

"cousin" modules, unreferenced in any of their common ancestors, should repeat the mnemonic definition in each such module.

5. Insert annotations or cross-referencing information so as to make a correspondence between specific detailed requirements or functional specifications and the current algorithm being documented when such information has not been included at a preceding level.

6. Explain the significance of each path following a decision, or the conditions assumed to be in effect at such points.

7. Include statements that provide rationale, assumptions, or other clarifying explanations of the algorithm as needed to lend meaning and readability to the text. Describing the intended significance of an action (such as, for example, setting or testing a flag) can save a reader much time in understanding what that algorithm is supposed to do. It is important to provide such information for every loop, stating what assumptions are valid during each iteration (i.e., the loop invariant).

12.7.4 Rules for Documenting Data Structures and Resource Access Requirements

Documentation of the data-structure development hierarchy, or other resource-access requirement hierarchies, is a dynamic process that requires visibility and interfacing, usually across wide segments of the development project. The evolution of a data structure design, for example, can be maintained in a Data Structure Definition Table (DSDT) in the Software Development Library or Project Notebook. When the definitions are complete, they are converted into formal descriptions in the SSD.

The rules of this section are aimed at making the concurrent documentation of such evolution compatible with later more formal inclusion in the SSD. Consult Section 12.7.1, Rules 7-9 for information concerning table content.

1. Maintain the current level of hierarchic detail for data structures or other resource access requirements in the most flexible and visible form available.

The use of 7-1/2 × 12-1/2 cm (3 × 5 in.) cards or pages in a loose-leaf notebook (each table beginning a new page or card) or computer files maintained by the SDL permits good expansion and revision capability.

2. Make an entry into the DSDT for each data structure or resource requirement referred to in general terms, or in any way other than its name. Later, augment the DSDT to state the actual structure name.

3. Strive to make tables provide explicit specifications, or else give references to such material elsewhere, either in the SSD or in external documents. Organize DSDT entries in alphabetic order by mnemonic name for ease in locating structures referred to. An example of a DSDT entry appears in Figure 12-26.

4. Include graphical displays and narrative descriptions of data structures and other resource access definitions. These should not only describe the composition and format of the data structure or other resource accesses, but should also describe the operations (or level of access) performed.

12.7.5 Rules for Documenting Structured-English Procedural Specifications

Besides flowcharts and accompanying narratives, there are other techniques to design and display procedures in a two-dimensional structured form and explain what is going on. I mentioned the use of CRISP-PDL in Section 7.4 as a simple, effective procedure design language. This section provides a few rules for using such a technique either as augmentation or as an alternate to the material in Sections 12.7.2 and 12.7.3. If used as an alternate, then CRISP-PDL should conform to the

SEGNO	SEGment number (NO), flag variable (range: 3-9) first assigned in SYSUP (module 2). Value specifies the configuration currently active, or next to be activated after configuration by USWAP (module 1J1), as detailed in the table below. SEGNO is active throughout the entire program except for the subprogram EXIT (module 9).
SEGNO = ?	Configure for
3	SYSIZL
4	PARSE
5	RUNIZL
6	RUN
7	BATCHC
8	BATCHR
9	EXIT

Figure 12-26. Example of a Data Structure Definition Table entry

discipline of both of these sections, as appropriate. As a preliminary design or look-ahead tool, the rigors may be more relaxed. A skeleton example is shown in Figure 12-27, and an SSD entry appears in Figure 12-28.

1. Structure the specification of program algorithms into a hierarchic, top-down syntax using the control language of Appendix G superimposed on simple English language constructions. Use indentations and cosmetics to display the nesting structure as indicated.

2. Limit each such specification to one page by inventing named procedural subspecifications for expansion at the next hierarchic level. These named subspecifications will be referred to as *striped modules*, just as if they appeared on flowcharts.

3. Begin each striped module algorithmic description with its name, identification number, and date of latest change.

4. Affix to each module a configuration-control box for concurring signatures (or initials) and dates of the module designer, design verifier, and project manager, or his designate.

5. Prior to the description of the algorithm of the current module, provide a comment block that states all assumptions on inputs, common data, etc., used by this module and visible in the algorithm. Describe the module function and its outputs, actions, and constraints.

6. Number each step in a procedure in numerical order from the top down, except for perhaps ENDIF, ENDCASES, JOIN, etc., which represent collecting nodes at the end of flowcharted structures. Affix this number to the statement beginning at the left margin and preceded by a period (see Figure 12-27).

7. Assign to each *subprogram* striped module a unique Dewey-decimal identification code. This code is made by concatenating the step number, where that subprogram is invoked, with the current-module identification code.

Thus, for example, if step 3 of module 1.4.6 invokes a subprogram, that subprogram is given the identification code 1.4.6.3. Note, also, the use of the virgule in Figure 12-27 to rename subprogram 1.8 as subprogram 3.

8. Assign to each *subroutine* striped module a unique alphanumeric code as its level-1 cross-referencing symbol. Annotate each procedural step *n* where a subroutine is called by placing the subroutine number *Sm* following the step number, separated by a virgule, as *n/Sm*.

Module 1
CRISP-X
5 Feb 75
page 1 of 1

PROGRAM: CRISP Processor

```

<*This module is the top-level preliminary design
<*of a CRISP translator for an arbitrary, as-yet
<*unspecified unstructured lower-level language, X.
<*The characteristics of this translator are set
<*forth in Appendix G of this text. No coding
<*language for this program has yet been chosen,
<*and codability of this design has not specifically
<*been considered. This documentation is for
<*architectual studies only.

.1      do Initialize the processor
.2      loop
.3          | do Accept control parameters
.4          | do Configure as necessary
.5          | if (control says EDIT mode)
.6/2      | : do perform source editing <*Note: edit
          | : <* functions are not included in Appendix
          | : <* G; this stub is a hook into an
          | : <* envisioned later capability*>
.7          | :->(control says PDL mode)
.8/3      | : do Cosmetize and list the program descriptions
.9          | :->(control says FLOW mode)
.10/4     | : do Draw flowcharts of the program descriptions
.11        | :->(control says X-translate mode)
.12/5     | : do Translate source descriptions into
          | : language X
          | : .. endif <* control option may skip all the above*>
.13        | : .. repeat unless (control says go on)
.14/6     | if (control says load-and-go) do Link target code
          | : to its compiler, compiler output to loaders, and
          | : .. tell loader to initiate execution of job.
.15        do Clean up required for exit
          endprogram

```

Figure 12-27. A conceptual level-1 design description of a CRISP preprocessor using CRISP-PDL-like language (several features that will be required later, such as error response and recovery, have been neglected at this stage of design; as given, this description probably represents Class C or D preliminary, or "look-ahead," design documentation, not suitable for formal specifications)

Module 1
CRISPFLOW
23 Sept 76
page 1 of 1

5(1) CRISPFLOW Detailed Design: Main Program Hierarchy

```

PROGRAM: CRISPFLOW <* 23 Sept 76 *>                                MOD# 1
    <* This is the top level diagram of the CRISPFLOW
    <* processor. See Section 2 of this software speci-
    <* fication document for program description standards
    <* and conventions, Section 3 for environment and
    <* interfaces, Section 4 for functional specifications
    <* including I/O, and Section 5.0 for an overview of
    <* processing and supporting data structures.

.1      DO CRISPFLOW_ INIT <* Declare and initialize all global
    <* data structures*>
.2      DO OPEN_IO_MEDIA <* Open necessary I/O and set options*>
.3      Set EOS = %FALSE <* end-of-source flag*>,
    ERRFLG = %FALSE <* processing condition flag*>
.4/S1   CALL GETLINE (INBUFFER, EOS) <* Set EOS true if no line left
    <* in source medium*>
.5      LOOP WHILE (EOS = %FALSE) <* until end of source input*>
.6      ! DO PARSE <* Bring in module and build TREE*>
    ! <* Output an error message and set ERRFLG true
    ! <* if a module cannot be flowcharted as per
    ! <* section 4 of this SSD*>
.7      ! IF (ERRFLG = %FALSE) <* If module was acceptable*>
.8      ! : DO LAYOUT <* allocate sizes and coordinates*>
.9      ! : DO DRAW <* Draw the flowchart*>
    ! :... ENDIF <* otherwise, ignore the module*>
.10/S1  ! CALL GETLINE (INBUFFER, EOS) <* get next input line
    !... REPEAT
.11     PRINT "FLOWCHARTING COMPLETE"
.12     DO CLOSE_IO_MEDIA <* Clean up for termination*>
ENDPROGRAM
  
```

Figure 12-28. A CRISP-PDL description of the top-level specification for the CRISPFLOW processor described in Appendix G

Thus, for example, if step 3 of module 1.4.5 calls a subroutine P6 named PLOT, then

```
" 3/P6 CALL PLOT(xdata, ydata)"
```

may appear as the procedural step description.

9. If internally defined function calls occur in a step, attach their level-1 number codes, as in rule above, to the step number as well, as room permits.

For example, if RIGHT is a function numbered N5, then

```
" 3/N5 SET TOKEN=RIGHT (TOKEN)"
```

may appear as a procedural step.

10. Make all conditional tests explicit (i.e., no striped decision modules). Annotate the steps corresponding to test outcomes with meaningful explanations of that outcome. Note that the description in Figure 12-27 has not adhered to this rule, being a look-ahead design. Figure 12-28, however, being an SSD entry, has conformed.

11. Do not include irrelevant or extraneous information in describing procedural steps. Be brief, using references to, rather than repeating, information appearing in parent modules or in auxiliary tables. No functional information given in a parent module need be repeated, except as needed to

- a. Identify which subfunctions belong to which steps in the current algorithm.
- b. Detail a given function into subfunctions.
- c. Clarify the algorithm (for example, by giving an explanation of the meaning of a certain condition that has caused entry to this module).

12. Use the *imperative mood* to specify actions taken within a module, and use *single-spaced text* to describe steps. Double space between steps, if desired, to group sets of steps for readability.

13. Use descriptive or mnemonic names for all data structures referenced. Provide the mnemonic derivation of each data structure name appearing in a module at least once (at its first top-down appearance). Names appearing in "cousin" modules of the tier chart, unreferenced in one of their common ancestors, should repeat the mnemonic derivation in each submodule.

14. Insert annotations or cross-referencing data so as to make a correspondence between detailed requirements or functional specifications and the algorithm being documented, when such data has not been included in the description of that module at a preceding level.

15. Include statements that provide rationale, assumptions, or other clarifying explanations of the algorithm steps as needed to lend meaning and readability to the text.

16. Provide cross-references for each subroutine which locates all calls to that subroutine. This material is used to aid in identifying, in the event changes are made, where side effects may be likely to occur.

12.8 RULES FOR THE SOFTWARE DEVELOPMENT LIBRARY

1. Make the SDL the central repository for all textual and graphic design material, be it working-level, look-ahead, or approved documentation.

2. Accept design modules into project control only from the top down. Keep a tier-chart log of module status which includes:

- a. Date of submittal.
- b. Date of acceptance.
- c. History of design changes or proposed changes.
- d. Assigned phase of development.

3. Publish regular reports regarding project status that reveal progress relative to the work breakdown structure, modules completed during the reporting period, the cumulative number of total modules completed, and the number of identified stubs at future levels which are to be yet supplied. Flag modules with apparent discrepancies, such as those having erroneous module numbers or names, those being supplied or worked on in the wrong development phase, and those with recognized anomalous behavior.

4. Distribute copies of design materials to team members upon request. Do not permit master copies of approved modules to be altered without approval of the project manager. The SDL keeps the official master copy in all cases.

5. Maintain a file of standards waivers, and provide periodic summaries of waiver activity to the project manager.

6. Disseminate standards information and materials within the project, and train new employees in the use of these standards.

7. Use available automatic or computer-aided means for the creation, update, and monitoring of design documentation. Implement policies

whereby team members may interact openly with the design base, but which discourage team members from keeping their efforts invisible or private.

In the Chief Programmer Team concept, for example, the programming secretary enters all information, coding, and testing into the computer; no others are permitted access to the computer. Such a policy may be necessary, in austere cases, merely to make programming a public practice.

8. Maintain a file of all action items levied by design reviews, with the current disposition of each.

9. Audit all documentation submitted for approval for adherence to standards.

12.9 SUMMARY

This chapter has put forth standard disciplines that encourage software design as a hierarchic layering of detail, both in the invention of control-flow algorithms and data structure abstractions, to focus attention to relevant details, and to hide irrelevant details, at each point in the design synthesis. I have tried not to orient these disciplines toward any particular programming language, existing or envisioned, but toward the thought processes during the design process, and toward documenting the results of that process in a way that is sufficient and useful throughout the entire software life cycle.

Further discussions on the design medium and automatic aids to design may be found in Chapter 17 and Appendix G. Samples of the documentation styles appear in Appendix L.

XIII. PROGRAM CODING STANDARDS

Once the program algorithms have all been specified, the data structures all defined and laid out, and the I/O interfaces all fixed, what essentially remains is the design of code to implement the "unstriped modules" of the SSD and linkages to and from striped modules. The activity of concurrent coding, however, requires a bit more than mere rote translation of the design specification into statements of executable code, modularly programmed to conform to the hierarchic design levels. In practical application, there are usually many accommodations that must be made before the coding can even get underway. Generally speaking, the methods here recommend that code be generated in an execution sequence; for example, job control code first, then linkage editor code, then module linking code, and, finally, code for the design specifications.

This chapter presents a set of rules for producing and documenting only the code specified in the SSD. These rules are based on the premise that noticeably more efficient programs result more from design-level considerations than from coding-level considerations, and that faithful coding of the design is of paramount importance.

13.1 RULES FOR CODING STRUCTURED PROGRAMS

At the onset of coding, all data structures and other design mechanisms will have been described, at least to the degree required, in a set of self-contained, cross-referenced, documented specifications (the SSD). In the rules that follow in this section, I have assumed that one is coding from flowcharts and narrative, although one could as easily be coding from CRISP-PDL specifications. (In fact, as will be seen in Chapter 17, the dividing line between such "design specifications" and "coding" in my "standard production system" is somewhat artificial because of this.)

Recall that striped flowchart symbols at one tier of the design were expanded into procedures at later levels, and that unstriped symbols were constrained to be codable without functional ambiguity—any method of coding such a submodule, as long as the code performs as specified in its unstriped description, must work.

Let me make a distinction at this point, and for the remainder of the chapter, between a "code module" and a "code submodule," for the purposes of describing the program code. By a *code module*, I shall mean all of the code that corresponds to a "flowchart" or a flowchart symbol and all its hierarchic descendent striped submodules, including any subroutines that perhaps may be used totally within the code module, and not outside. By a *code submodule*, I shall refer only to the code that appears explicitly for coding a single flowchart. Whenever a striped symbol appears on a given flowchart, then only the linkage (if any) to the remainder of the code for that striped symbol will be counted as part of the code submodule.

Thus, for example, the code submodule MBASIC™ depicted in Figure 13-1 possesses less than one page of code; the entire MBASIC™ code module, however, consists of tens of thousands of code instructions.

Let me also distinguish code modules and submodules from "compile modules." A *compile module* is a set or subset of code modules and/or submodules that are compiled together as a unit for any reason, such as to provide a level of access to a data structure, to segment a program into overlays, and so forth. The grouping of code submodules into compile modules is part of the code design, and much of it, if not all, will be dictated by the program specification.

By definition, the term, "coded," as used here, means that the unit referred to has been translated into a computer language, exists in a machine readable medium, and contains no syntax errors.

The following rules are guidelines for the generation of code modules and submodules for structured designs.

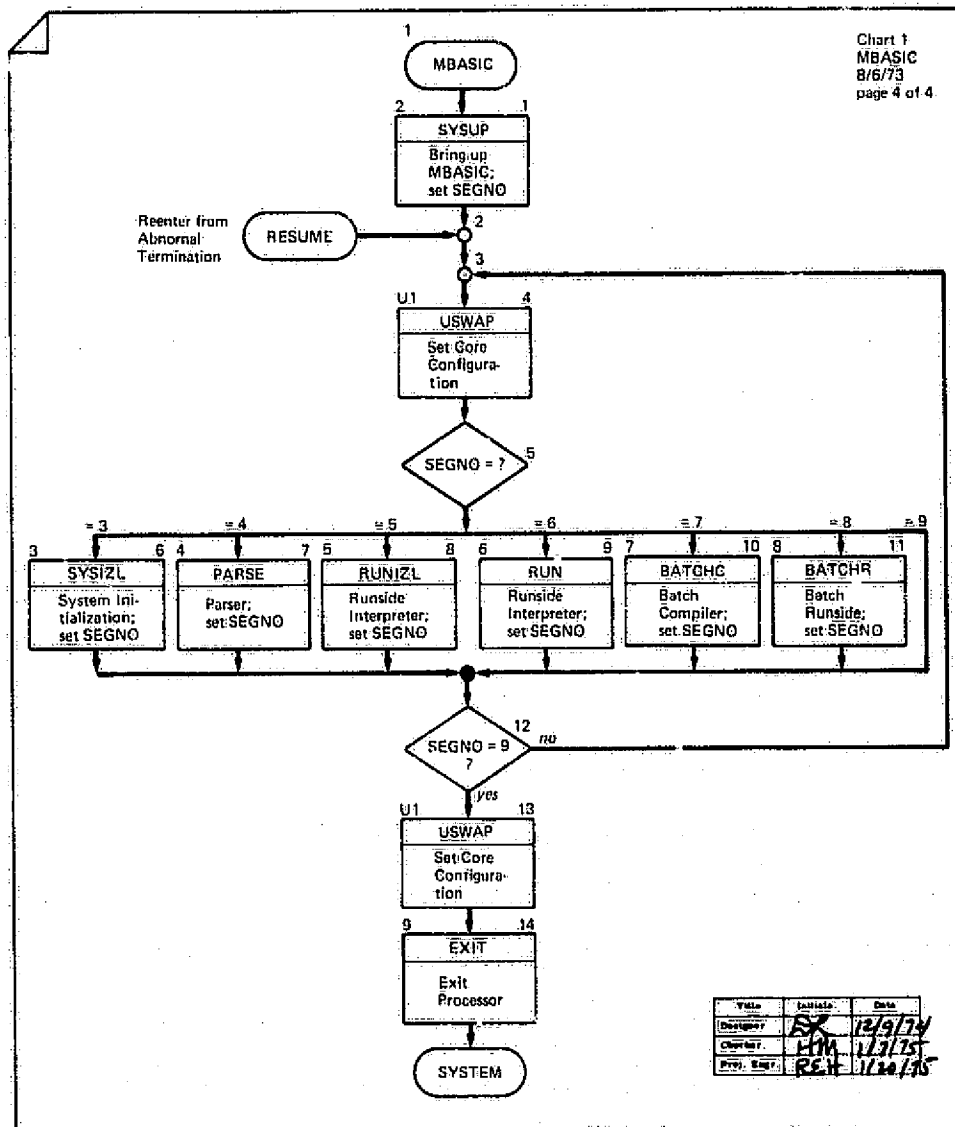
1. Become thoroughly familiar with the overall design philosophy and conventions before coding begins.
2. Establish project-wide coding standards for style. Use macros or other style conventions only when these are accepted and used as project standards. Conform all situations where such standards apply ruthlessly to the standard, with deviations only on an approved waiver basis.
3. Make coding a direct translation of the flowchart. Code programs into a format conforming to and reflecting the structured, hierarchic, top-down, modular design.

Normally, code unstriped symbols as in-line code, and code striped symbols (references to procedures) as linkages to separate procedure modules (unless timing or memory is critical and such linkages are significant—see Rule 6, below). Figure 13-1 illustrates the one-to-one correspondence between flowchart and code for an assembly-language submodule.

4. Set standard invoking (or calling) sequences for subprograms, subroutines, and abortive branches, especially if these involve recursive or reentrant linkages. Prove that such techniques work independently of the program module being developed.

Code each flowchart as a separate code submodule or macro procedure using the most efficient linking method available in the programming language. Specifically, if the linking method makes use of a stack to hold arguments or return linkages, and if abnormal terminations lead to reinitialization of the program, make sure that the linkage-stack is also reinitialized (notice that this is done in Figure 13-1). When linking modules using the mechanisms provided by a higher level language, it may be necessary to know how these mechanisms are implemented (viz., whether a stack is used) in order to provide the proper reinitializations required by abnormal termination.

5. Limit coding to clear, concise code that will be easy to check, debug, and modify. Avoid coding-level optimizations that decrease clarity. If more optimal code is needed, then clearly annotate the listing so that the code remains an accurate representation of the procedure specification. Do not



```

      TITLE      MBASIC
      *****
      ;          MBASIC
      ;
      ;          1
      ;          RBH  3/20/75
      ;          RCT  3/21/75
      ;          RBH  3/21/75
      ;          *****
      TWOSEG;
      RELOC  400000;
      SEARCH MACROS;
      SEARCH SYMBOL;
      SALL;

      MBASIC: MOVNI  LPTR,MAXL;          .0 NECESSARY CODE TO
      HRLZ      LPTR,LPTR;              INITIALIZE MODULE LINKING PTR
      HRR1      LPTR,LSTACK##-1;        SET POINTER TO (-MAXL,LSTACK-1)
      DO        SYSUP##;                .1/2
      RESUME::MOVNI LPTR,MAXL;          .2 REINIT MODULE LINKING
      HRLZ      LPTR,LPTR;              PTR UPON ERROR: SET
      HRR1      LPTR,LSTACK-1;          POINTER TO (-MAXL,LSTACK-1)
      MBA.03::;                          .3
      DO        USWAP##;                .4/U1
      MOVE      R1,SEGN0##;
      XCT       MBA.T1-3(R1);           .5
      JRST      MBA.12;
      MBA.T1: DO  SYSIZL##;              .6/3
      DO        PARSE##;                .7/4
      DO        RUNIZL##;              .8/5
      DO        RUN##;                 .9/6
      DO        BATCHC##;              .10/7
      DO        BATCHR##;              .11/8
      NOP;
      MBA.12: MOVEI  R1,9;               .12
      CAME        R1,SEGN0;
      JRST      MBA.03;
      DO        USWAP;                 .13/U1
      DO        EXIT##;               .14/9
      END       MBASIC;

```

Figure 13-1. Correspondence between flowchart and code for a module MBASICTM/1 (the language is PDP-10 MACRO-10 (assembly language); the preamble prior to the entry label announces to the assembler that this is a reentrant program and defines the symbol and macro definition files; DO is a striped-module-linkage macro)

use obscure, undocumented, or unmaintained features of some instruction sets or operating systems.

6. Code procedure instructions so as to remain fixed during execution; do not write code that modifies itself. Sharing memory by core-swapping is permissible, however.

7. Code striped-module subprograms or subroutines as in-line procedures only whenever (a) execution time or storage is a critical parameter, (b) the execution time or storage for the calling/return sequence is non-negligible, and (c) the function is not used so many times in the program that repetitions of in-line code would create a burden on the memory requirements. Use macros to define such in-line procedures, when available. Otherwise, program such functions as in Rule 3, above.

8. Use the same symbolic names for procedure entry points, variable names, etc., as appear in the program procedural design, if permitted by the implementation language. Clearly annotate listings when alternate names or labels have had to be used, and create a glossary of such correspondences.

9. For operations that act on data structures, pass only the field or fields necessary for that operation to accomplish its function, rather than moving the whole structure or substructure.

10. If correctness assertions have been supplied in the procedural specification, then insert code to check these assertions at runtime. If possible, make such code a compile-time option that can be overridden or removed from the object program after checkout is complete; however, do not remove this code from the source code, as later modifications may require it in further checkout.

11. Insofar as possible, put all implementation-dependent parameters or compile-parameter options together in one place in the code. Identify these as such, and give a prescription for changing them should the parameters require alteration.

12. When using a programming language which permits several data structure names to refer to the same or overlapping storage structures (e.g., by way of the EQUIVALENCE statement in FORTRAN), confine the configurations so that no more than one of the data structure names is active at one time. Enter the names of each data structure sharing the same storage appropriately in the code documentation. Do not reference the same data structure within its active range by two different names unless the conventions are so documented as coding standards and enhance the readability of the code.

13. Code the flowchart boxes such that the code for nodes of the flowchart appear in the same order as a pre-order-traverse list of the graph nodes. (This rule assumes the flowchart has been drawn with branches in case-order from the left.) The step numbers then fall in sequence down the page (see Figure 13-1).

Figure 13-2. A CRISP program segment that uses macros to rename specified data structure names as reusable stack locations

Take care to assure that external system subroutines which have both normal and paranormal or abnormal returns are coded to conform with the prescribed flowchart structures (see Figure 13-3).

14. When coding in a language requiring jumps to achieve structure, establish named labels for all locations branched to, such as decision collecting nodes, loop collecting nodes, and procedure entries. Set a standard for generating these names that does not conflict with names of variables used in the design. Do not use "current location $\pm n$ " as branch targets even if permitted in the coding language.

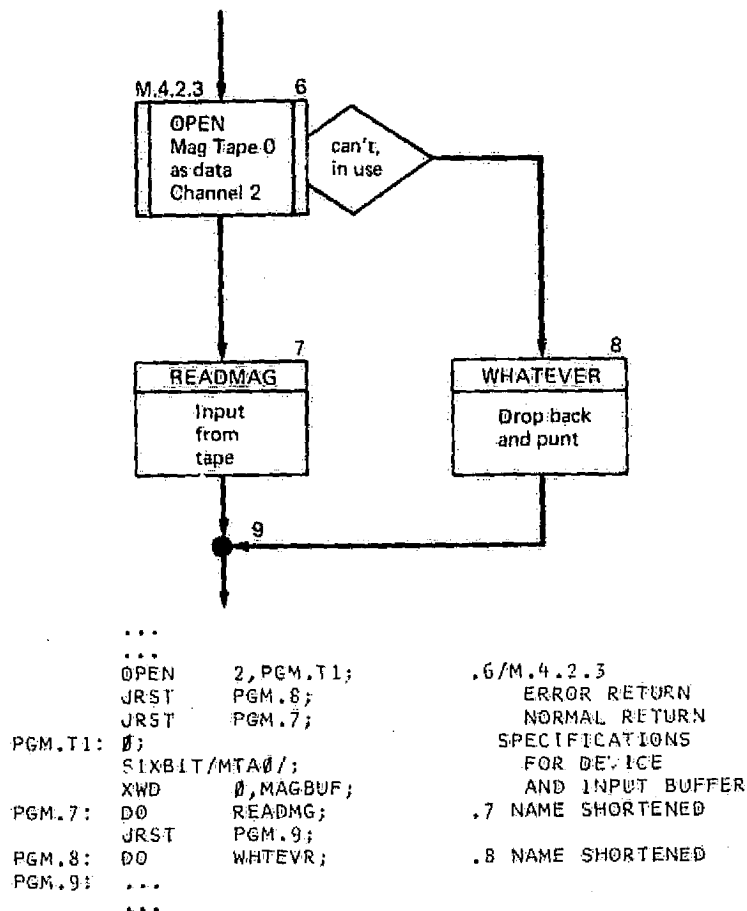


Figure 13-3. Usage of a system routine that has "normal" and "error return" points (the code is PDP-10 assembly language; the cross-reference in box 6 is to the Monitor manual, Section 4.2.3; DO is the striped-module-linkage macro)

15. Strive to localize the scope of variables within each code module when supported by the programming language. Pass data to subroutines normally as arguments; when arguments prove infeasible, then permit links to global or common pools. Do not permit content coupling between separate compile modules (see Section 4.4).

16. Define named constants and compiler parameters to indicate purpose, rather than value. Use separate constants (parameter names) for different purposes, even if some of these have the same values, for later modifiability. Avoid the use of "magic numbers," or numbers whose meanings are not implied by their value.

17. Use literal values only if these are truly constants in the problem. Literals should not be used to address data structures by making use of assumed structural formats, especially if this will limit the extendability, modifiability, or flexibility of the code being written.

For example, if the third word of an array, `RECORD`, contains a field to be accessed, do not use such devices as "GET `RECORD(3)`," because the `RECORD` format is likely to change eventually.

18. Set standards for representation of primitive and extended data types. For example, set a common convention for *true* and *false* tests throughout the program, if not already a part of the language syntax.

19. Set standards that relate preferred methods for coding to enhance speed or memory usage, such as indexed vs. indirect addressing, register-to-register vs. register-to-memory operations, byte access methods, coding the normal branch as the higher-speed conditional branch, etc.

20. Avoid passing program labels as arguments. Such practice implies the intention to use that label as a control branch target, and assessment of control correctness is thereby made more difficult.

13.2 RULES FOR CODING STRUCTURED REAL-TIME PROGRAMS

The structure imposed upon real-time programs by design specifications is modular partition into sequential activities which can be coded separately and then combined for execution in a way that should allow for correctness to be assessed. The following rules for coding real-time programs augment those given in the previous section.

1. Determine methods to be used and establish standards to provide arbitration among resources as required in the design. Code and validate such means, as a general rule, before coding any part of the program making use of these features. (Note: top-down design is still in effect here; some bottom-up coding may be required, however).

2. Locate the code bodies for concurrent processes appearing within the same program in conformance with the same standards set for non-real-time program segments in Rule 13 of Section 13.1. Make the code conform to procedural specifications in a one-to-one identifiable (and QA auditable) way, step-by-step, module-for-module.

Consult Appendix G for FORK-JOIN and WHEN structures. If program-operated dedicated traps appear, code these to conform with the AT structure.

3. If more than one user can concurrently operate a given process (program or subroutine), code all module and submodule linkages involved so as to enable reentrant use. Make provision for each caller to have his own (protected, if possible) separate data workspace for data accessed by the process.

4. If reentrant code is being written, get the reentrant mechanism working before coding any of the design of the reentrant elements.

5. Code the procedural design, then, from the top down in testable phases. That is, conform coding to match development testing phases.

6. Code stubs for real-time checkout that consume the proper durations, as well as recording trace information for correctness assessment.

13.3 RULES FOR DOCUMENTING STRUCTURED CODE

The rules for documenting the program code are formulated so as to make the code a readable extension of the programming specification. The rules are not in the direction of self-documentation, but just the opposite! The top-down development process has gone to a lot of pains to make sure that program specifications, such as narrative and flowcharts (or equivalent), provide all the information needed to understand the program, without references to code listings. Having listings readable by themselves negates the life-cycle value of the SSD; the code becomes the maintenance-analysis medium, rather than the SSD. Changes to the code don't always get inserted back in the design documentation when this happens. As a result, the SSD is soon out of date, a worthless, expensive document.

Such a philosophy as outlined by the rules that follow is necessary to avert the possibility that a program can be maintained only at the code level. It, moreover, encourages the use of computer-based media for holding and displaying both design specifications and the code. In Chapter 17, I shall give a special interpretation of these rules so as to keep flowchart, narrative, code, and annotations all together in one source.

1. Enter into a special, easily located part of the program documentation all general, program-wide coding conventions and standards that relate how flowchart specifications are coded. Make the set of conventions complete enough so that, except for special cases covered in the next rule, they are sufficient for a reader to assess that the code is a true translation of the program specifications.

2. Annotate the code corresponding to flowchart specifications with any special supporting information needed to understand how those specifications have been implemented into code. Such annotations explain each instance where an unusual or non-standard feature of the programming language is used and is material to ascertaining coding correctness.

For example, the FORTRAN statement " $I=N/2$ " assigns to an integer variable, *I*, the integer part of the division of the integer *N* by 2. If it is possible for *N* to be odd, and if program correctness actually depends on the implied (automatic) truncation, then the statement should be so annotated. As another example, the MBASICTM statement "COPY 'LINKFILE'" clears the current program workspace (but retains values assigned to variables), loads the program in the file named 'LINKFILE' and returns the MBASICTM processor to the command mode. If, however, the copied file contains a statement "GO TO linkstart" (where *linkstart* is a statement number in the 'LINKFILE' program), then there is an automatic run-initiation of the linked program that is not visible to one reading the program in which the COPY statement appears. In such cases, the COPY statement should be annotated so as to inform the reader that the current program will be terminated and cleared from workspace, that the copied program will be executed immediately after the COPY, and that any code subsequent to the COPY command in the current program will be ignored.

3. Do not repeat information appearing in the software specification. Instead, if such information is necessary to understand the coding, give a reference to the proper point in the specification. This practice makes the

code readability rely on the program specification and avoids duplication of documentation.

4. Conform annotations, whenever possible, to make use of any automatic program-checking facilities available, such as automatic flowcharting, automatic annotation, control-logic verification, and statistical testing.

5. Annotate source-language code listings to facilitate subprogram and subroutine identification, reference, and change control. Specifically, annotate the code for each striped module with a header containing its subprogram or subroutine name, its Dewey-decimal number, initials (or names) and dates of coding, peer concurrence, and testing, all in a consistent, uniform way.

6. Identify steps within modules by annotations that supply the same identifying numbers as they appear in the design specification. Figure 13-1 illustrates such annotation of an assembly-language program coded from a flowchart.

7. If a cross-reference number appears on a flowchart box, then supply this number to the box-reference code in the annotation for the code corresponding to that box.

For example, within a flowchart module "RUNIZL/5" (Initialization for RUN, chart #5), suppose there is a flowchart box numbered "1" that calls a utility subroutine "SETRIO" (SET up I/O for Run), which is numbered as U7. Then the beginning line of the code corresponding to the "DO SETRIO" on the flowchart has the identifying annotation ". 1/U7" (see Figure 13-4).

8. Indicate the program structured hierarchy by indenting lines of code, when permissible in the programming language, to reflect the levels of nesting of structures within other structures.

9. Annotate dummy stubs to be removed later. Use a special, standard, easily distinguishable series of symbols to identify such annotations. Identify any special debugging code similarly.

For example, suppose an MBASIC™ program has a table, TABLE, to be printed out in a test mode. Then a probe instruction might appear as ("!" starts the comment field)

```
61 PRINT TABLE ! TEST PROBE*****
```

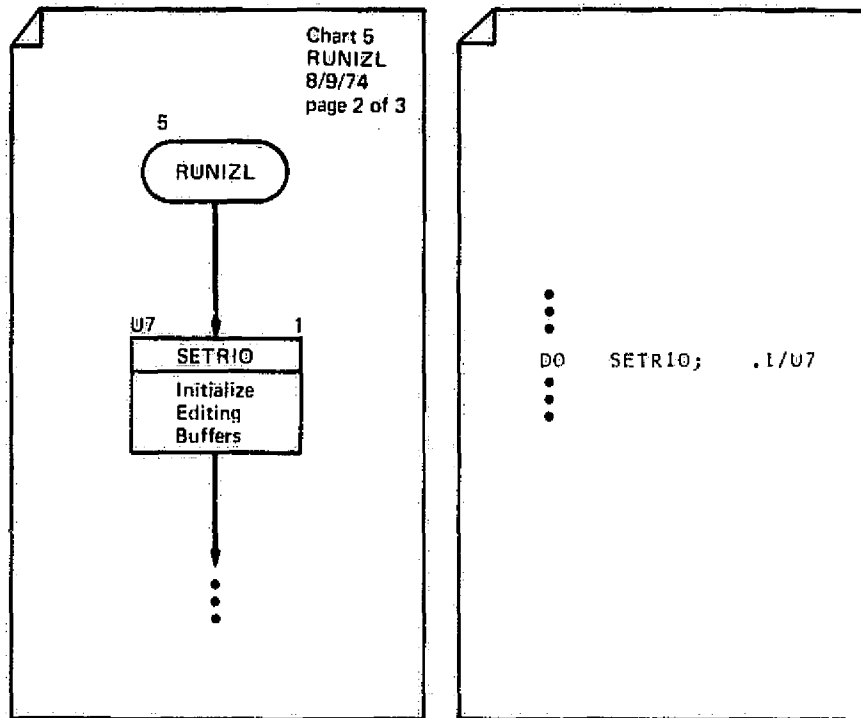


Figure 13-4. Flowchart and code listing for RUNIZL/5 module of MBASIC™/1 shown in Figure 13-1, illustrating the use of annotations to reference code to procedural specifications

10. If there are code-level assumptions upon entry or exit from a module, state these in a comment block between the module entry and the beginning of its compiled statements.

Examples of things that might appear in this section are memory protection levels required (read-only, write-only), reentrant or recursive, register contents and usage, etc.

11. If system utilities are used with compile or run-time options, give interfacing data: names of routines, calling methods, parameters and their meanings, function or service performed, requirements for calling, etc.

12. If the data structures defined in the SSD require further hierarchic detailing at the code level, then insert such information as annotations located with the data structure, properly referenced to the SSD. Record further hierarchic detailings of other resource access requirements in a similar manner.

13. Provide other auxiliary documentation as may be required to bridge the design to the code, such as:

- a. Cross-index lists (e.g., chart to file containing its coded form).
- b. Public features used (e.g., as index and system registers, buffers, etc.).
- c. Glossary of special variables, compiler parameters, flags, literals, and storage structures not easily defined by name or use context.
- d. Memory use map.
- e. Timing diagrams.
- f. Interrupt handling procedures and relationships.
- g. File, table, and data-set descriptions.
- h. Examples of input and associated output.
- i. Listing of special flags, pointers, and other indicators, together with their usage (which routines, areas or time of applicability, etc.).
- j. Commentary describing features of code that link performance to design documents.
- k. Lists of error conditions, codes and messages, cross-referenced to both design charts and the code itself.
- l. Restrictions on the use of code that is particularly sensitive to changes in the design (mainly in time and memory space, but also functional limits to subroutines, etc.).
- m. Data usage (e.g., shared vs. public files).
- n. Hardware or software constraints.
- o. Use of privileged instructions.
- p. Procedures for compilation, linkage edit, etc.

13.4 STANDARD PRODUCTION PROCEDURES

This section contains a set of guidelines by which the Software Development Library monitors and aids the production of the program during its evolution.

1. Maintain code, at its current state, in files available to all development team members on demand (see Figure 13-5). Establish a read-only "control copy," which includes only the approved code. The control copy does not contain any code that will not be a part of the final program.

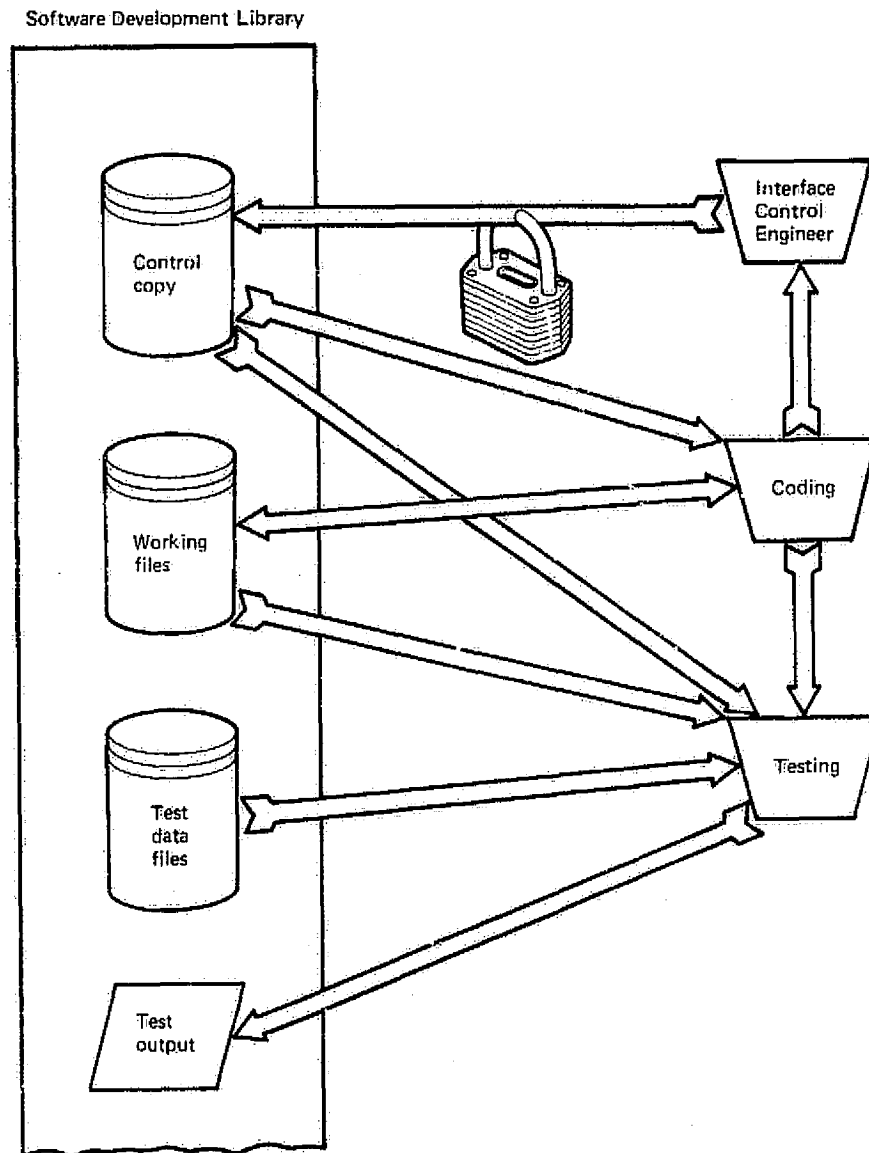


Figure 13-5. The Software Development Library, showing interaction among control copy, working copy, and other files with software team

2. Support the development coding of new modules and developmental testing so that such activities may take place only by supplementing and updating the control copy with the new code. Working copies may also contain dummy stubs, interspersed statements to collect or print trace information, etc. The control copy may be updated only upon proper authorization.

3. Maintain a tier chart or data base of the project-controlled code status. For each module, supply name, number, dates of submission and changes, and its state of completion.

Adopt a uniform notational scheme for annotating the chart, such as "S" for "stub," "P" for "preliminary," "L" for "look-ahead," "A" for "audited," "C" for "concurrent," "R" for "returned for rework," and "*" for "completed" (see Section 10.5.6 for an example).

4. Arrange to have modules within each loadable program segment of the control copy entered so that, when listed, they appear in their Dewey-decimal order. This procedure is an aid in locating code segments.

For example, suppose that program #1 has submodules 1.2, 1.4/S1 (i.e., Subroutine #1 appears in box 4), and 1.7. Then the listing order of the program at the completion of coding tier-1 is

```
1
1.2
1.7
S1
```

If 1.2 and S1 require no expansion, but 1.7 has submodules 1.7.3 and 1.7.5, then the listing at the end of the next tier would be in the order

```
1
1.2
1.7
1.7.3
1.7.5
S1
```

and so on.

5. Develop, document, and issue procedures by which team members can access, operate, and update the evolving program. Give specific log-on and

log-off procedures, as well as procedures for accessing the read-only files and building working copies.

6. Maintain a standard production configuration including compilers, editors, and other production software (Chapter 17), as well as manuals, operating procedures, etc.

7. Provide an archiving function for test data, dummy stubs, and test results.

8. Create a file naming and management policy for project files to ensure that program modules, data, and the like do not get confused.

For example, one could make compilable segments contain only submodules of a common ancestor module, and name the file by that ancestor. If several segments are merged for testing purposes, name according to the modules or phase being tested, and attach a test-assembly code number, TAn . Similarly, attach TCn to test code, TDn to test data. In each case, conform the version numeral n so that the assembly, data, etc., all agree.

13.5 SUMMARY

The standards for coding set forth in this chapter are somewhat unusual, in that they emphasize coding from a documented specification: once the program is coded and verified, the "as-built" specification becomes the paramount item for later sustaining effort. The code cannot stand alone—but conversely, the specification is detailed enough that one can read and understand it without reference to the code listings! The documentation is, therefore, of necessity kept current when any changes are made; program analysis, testing, and debugging are done primarily at the documentation level, not at the code level.

I do not want to imply by this, however, that specifications are necessarily going to be kept apart from the code. Indeed, as I shall discuss in Chapter 17, there is a way to integrate code and specifications into a single medium, or program base, if proper computer facilities are available.

PRECEDING PAGE BLANK NOT FOLDED
PRECEDING PAGE BLANK NOT FOLDED

XIV. DEVELOPMENT TESTING STANDARDS

Development testing forms the practical basis for assessing program correctness, and, for this reason, it could as well be called "correctness testing," or "validation." More often, it is just called "checkout." These tests are run within the development project as the program evolves as a reinforcing measure to assure the developers that the program, thus far, actually performs as specified. The guidelines set forth in this chapter provide an organized discipline toward accelerating this checkout process. At this point, let me summarize what has been required during the design and coding process for the purpose of assessing program correctness:

- a. Control-logic has been specified to such a level that will permit an assessment of correctness of the program flow on an individual module basis.
- b. The functional behavior of each module has been specified to that degree of detail that will permit an audit of the subfunctions of each module against its stated function, and will also permit a correctness evaluation to be made.

- c. Each module has been specified in sufficient detail to permit coding without functional ambiguity, as a unit, using linkages to striped modules or stubs.
- d. The code is a faithful translation of the design specifications.

Overall end-to-end and final acceptance testing for delivery is covered in the next chapter as a Quality Assurance function.

14.1 RULES FOR SPECIFYING DEVELOPMENT TESTS

The objective of this section is to set forth a set of basic rules that will increase confidence in program correctness by demonstration of its implemented features.

1. Devise a sequence of tests or test policy to validate the program at the current phase in accordance with the remaining rules of this section. For each test, decide which modules and program paths are to be exercised.

These tests should thoroughly exercise all error or overload recovery mechanisms. It is the role of testing to verify that such recovery modes have actually been provided for in the design and that they function effectively.

2. Identify for each test the set of needed striped modules not yet coded, and then design dummy stubs for such modules for testing purposes. The function of these stubs is threefold (Figure 14-1):

- a. To affirm that control flags (and their values) required by a submodule are actually available in the proper form to that submodule.
- b. To affirm that submodules which are specified to alter control flags may be simulated in a way sufficient to verify control correctness of their parent module.
- c. To provide dummy initializations or translations of data sufficient to verify the interfaces between all modules at the current phase. An "interface at the current phase" is defined as the set of all assumptions that modules within the current phase make about each other.

3. Prepare tests for each module added at the current phase using a test driver (the entire program up to this point may be used and is preferred, other things being equal). Such tests are to check for proper control by providing simulated input data or control flags sufficient to exercise *every*

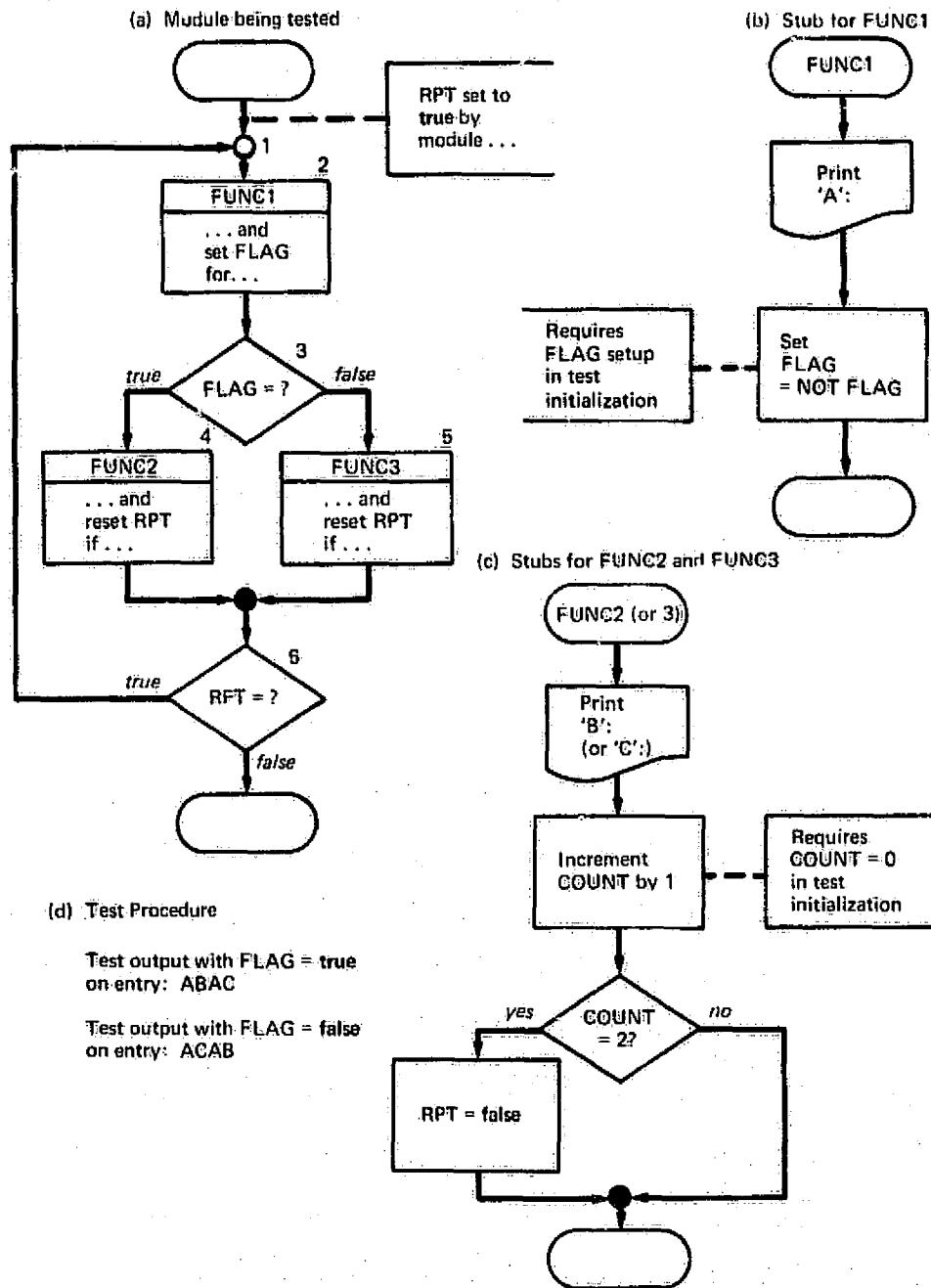


Figure 14-1. Testing each flowline using dummy stubs (printout shows coding complies with flowchart; functional correctness can be based on whether printout sequence and other trace information appears proper to compute module function)

flowline in the module. Such tests must demonstrate that the module has performed correctly and that the module interfaces within the current phase are consistent.

4. Use path monitors, trace printing, and other test-code features that will provide visible (and savable) evidence of the partial program correctness at the current phase.

For example, one may specify that all dummy stubs, upon execution, print out their module name, variables declared, variables changed, conditions checked, etc. Also, special execution counters can be inserted into each flowline to count the number of times each flowline is traversed during a given test.

5. Perform tests after the successful demonstration of Rule 3, above, which include extreme-value, permissible data; out-of-range, non-permissible data; and random data.

Design such test data principally from the user's or operator's viewability of the program, rather than as one who is acquainted with the internal algorithms. It is useful to design such tests using the user or operator manuals, in whatever state they exist at this point.

6. Define tests hierarchically, so that tests using dummy stubs can be rerun later (in greater detail) when the stubs are replaced.

For example, if a given set of data or conditions cause a dummy stub to be invoked in a test, then similar sets of data or conditions can be added at later phases to test the flowlines within the striped module replacing the stub.

7. Seek the simplest and easiest tests that yield a high confidence in program correctness.

For example, one may be able to test an indexed loop for operation at $k = 1, 2, \dots, n$ (k being the loop index) to ascertain that the loop is operating correctly for n much smaller than its actual later value; based on such tests, one may then, perhaps, be able to assess that the algorithm is correct for a general value of n , up to the actual value.

14.2 RULES FOR DEVELOPING TESTS FOR REAL-TIME PROGRAMS

Real-time, concurrent processes are harder to test than non-real-time programs because of their general asynchronous nature. However, the following guidelines are a beginning.

1. Identify the testable status in each phase of development and devise methods either to invoke such events in real-time, or else to simulate them in parametric time. Define tests to embody these methods, each of which introduces a minimum number of untried things.

2. Design dummy stubs for striped modules missing in the current phase that will permit real-time tests to operate. In particular, make sure that such stubs maintain program consistency (i.e., repeatability of results in a "practical sense," even if there are bugs).

3. Establish the ways processes can interact in terms of tests that can be performed to validate that interaction, and then input this test philosophy into the program design activity.

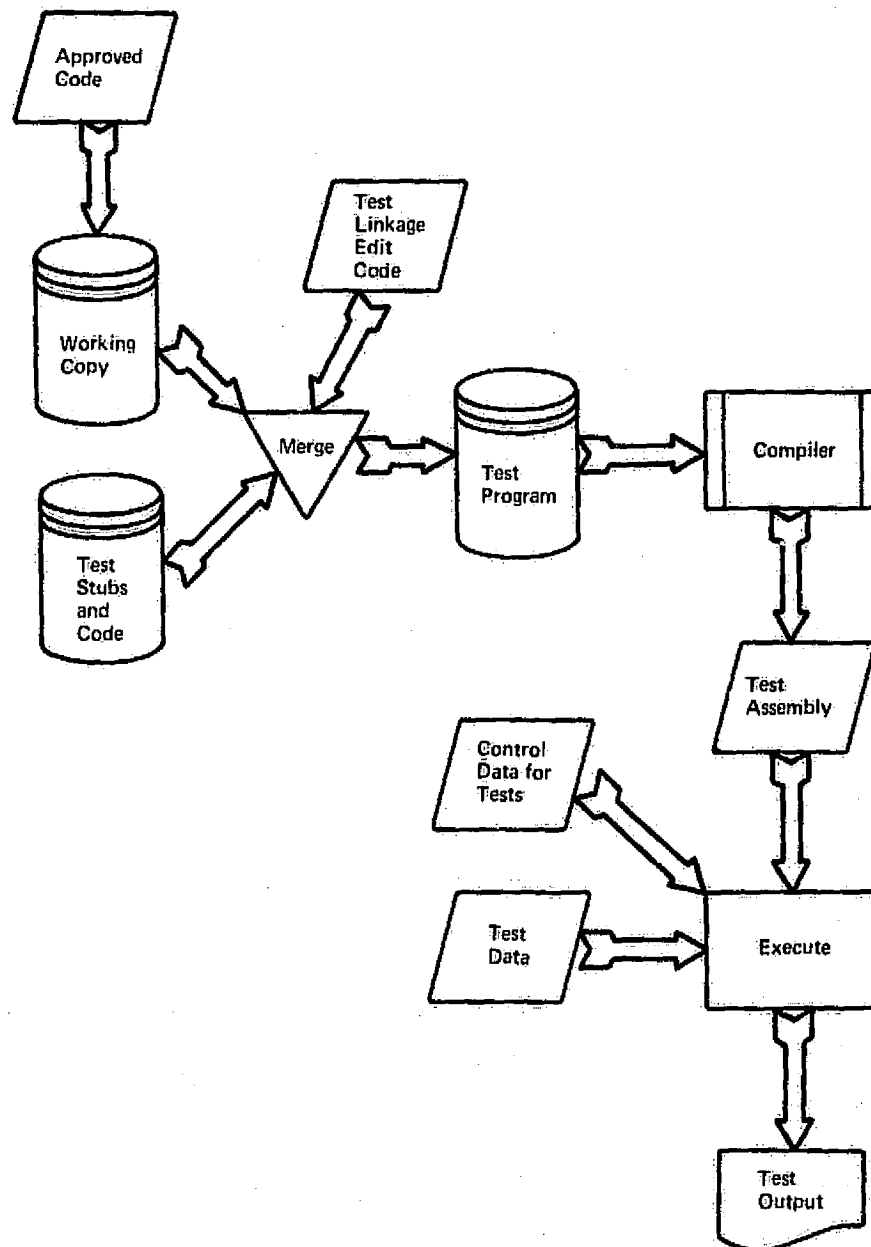
4. In addition to tests that traverse each program flowline, develop tests to exercise the program's response to dynamic conditions and to evaluate the program's resistance to deadlocks, thrashing, and missed deadlines.

5. Develop tests that include perturbations in hardware configuration, timing, or faults. Such tests are, perhaps, best done by simulating the hardware until consistency and correctness are verified.

6. Consider the use of hardware simulation to provide test cases that exercise the basic features and interactions of the software and stress the design criteria as well. Simulation is particularly effective in stressing the software in ways that are difficult to control on actual hardware.

14.3 RULES FOR ASSEMBLING AND PERFORMING TESTS

1. Append or merge dummy stubs and other test code with the "control copy" of the program in the SDL to form a scratch-file program on which development tests are to be run (Figure 14-2). Do not maintain a separate copy of the evolving program for testing that could later be "cleaned up" to become the final program.

**Figure 14-2. Assembling and running tests**

2. Establish and code linkage-editing procedures to collect the control copy, dummy stubs, and other test code into an executable program for each test. (In Figure 14-2, the test version of the program is denoted as "Test Assembly"; the linkage-edit code is identified as "Test Linkage Edit Code.")

3. Test real-time programs for consistency and then ascertain correctness.

14.4 RULES FOR CODING TEST ELEMENTS

In coding stubs and other dummy test code, it is worthwhile remembering that, while this dummy code may respond test data back to a module and its hierarchic ancestors, that data is not the actual data that the program will access during final operation. It is data supplied to verify logical control and *data-space control* functions only. Therefore, testing a module having dummy stubs succeeds only in testing the control aspects of that module relative to any data emanating from the stub. The data design is not completely verified until the actual data structures are actually accessed in their specified manner. Nevertheless, stub tests do contribute to data structural correctness.

1. Code test stubs in accordance with any specifications for path monitors, trace printing, etc., as may have been prescribed.

2. Keep the dummy code simple. Do not try to perform the functions the actual module must perform. Rather, simulate the interface needed for the tests specified only. Remember, stubs are discarded once the actual module has been coded.

For example, suppose a module specified to "accept commands from a terminal, translate these into the setting of a control flag, as specified in Table 4.8.T4, and return this value to the program" is to be dummied. It is then sufficient for the stub only to return the values of the flag in some simple, test-controlled way, in order to test the effect of command inputs on the remainder of the program (Figure 14-3). The stub need not recognize the input commands themselves.

3. Code stubs primarily to simulate control and data operations for the current test(s). Do not create, test for, nor alter data that is outside the scope of current test(s). That is, do not code irrelevant actions into stubs.

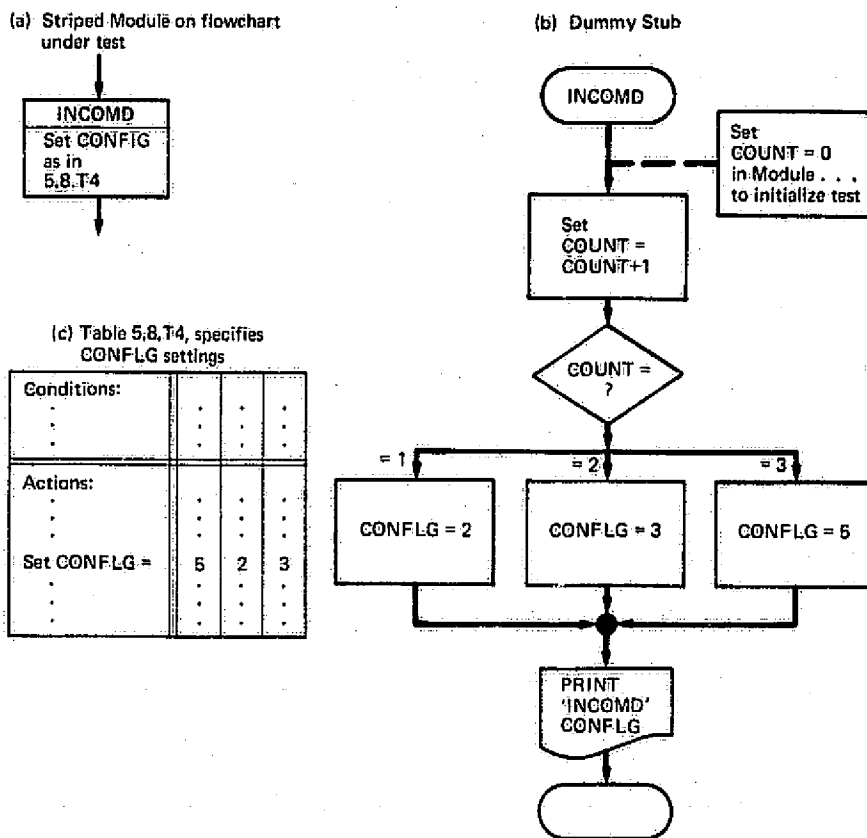


Figure 14-3. Simple example of a dummy stub that merely sequences through control flag (CONFLG) values to test control logic of parent flowchart

4. Code several separate, simpler tests rather than one large test, if this will speed the total job of stub coding and program testing.

5. Insert trace-monitor code in paths (other than dummy stubs) on an as-needed basis for checkout.

Trace-monitor code is code inserted into program paths to verify that the program during execution went through that path. In some instances, it is wise to "breakpoint" (test stop) a module at its entry to set appropriate data values, and, on exit, to query the exit status.

6. Perform tests, first, to prove that the code matches the flowchart (or other procedural specification); then, examine test output to assess program design and operational correctness.

In some instances, due to the great number of dummy stubs which may be present at that time, it may be advantageous to delay the design or operational correctness testing until a significant number of the dummy stubs have been replaced by the modules they represent. Any questionable design items noticed should, of course, be marked for later evaluation in the overall system.

14.5 RULES FOR DOCUMENTING DEVELOPMENT-TEST SPECIFICATIONS

1. Insert test and dummy stub design guidelines into the SSD Test Specification section. This material should include, for each test:
 - a. The modules being tested, the phase, group, version, etc.
 - b. The purpose of the test (i.e., which external specifications are being verified).
 - c. Test inputs.
 - d. Test procedures, policies, or guidelines.
 - e. Outputs to be achieved.
2. Specify test code, test data, and test procedures in detail sufficient for coding and testing, to the extent that any test code written, or any test data used, or any tests judged to be in compliance with the specification, should then yield the same high confidence in program correctness.
3. Specify how correctness of the modules under test is to be inferred from the test output. For each test, identify which functional specifications are being validated, or give a rule for determining which functional specifications will have been validated based on examination of the output.

14.6 RULES FOR DOCUMENTING TEST RESULTS

Development-test documentation consists of an evolving Software Test Report (see Appendix J), which summarizes each test, the modules involved, and the results achieved, and locates the output in the project archives. The following rules help organize this documentation for completeness and readability.

1. Write a simple description of each of the milestone tests performed, and insert this in the Software Test Report. Identify the modules in the

current phase that are being verified by these tests; it is permissible to designate complete design phases as entities, as "all of phase 3, plus modules...."

2. Archive code listings of each of the modules being tested and each of the dummy stubs. If a module or dummy stub used in current testing is the same as that used in a previous phase, it need not be relisted if properly referenced, as "The dummy for PARSE is the same as that listed in Section... of this Report."

3. Maintain a "Development Test Directory" as part of the production log in the Project Notebook, which lists all major modules and gives cross-references into the archives of tests.

4. In cases where test specifications or criteria are broadly stated, report how correctness of the modules was inferred from the test output. For each such development test, specify which functional specifications are being validated, and the extent, if only a partial validation.

5. Collect all test output for storage in the project archives. Arrange the test output in the archives so as to be locatable using the development test directory.

6. Log in the Project Notebook and summarize in the Software Test Report all failures, ambiguities, incorrect actions, etc., for each test.

7. Maintain a file of "Discrepancy Reports," into which are accumulated all of the problem reports related to the design, coding, and testing, up to the current phase.

Include in such reports the category of problem, such as "control logic error," "data extraction error," "undefined variable required," "destroyed variable needed by another module," etc. History of this sort is needed to refine standards and improve future estimates of project activity.

14.7 RULES FOR THE SOFTWARE DEVELOPMENT LIBRARY

1. Establish a standard file-naming and management policy for test assemblies, test data files, dummy stub files, test code files, scratch files, etc. (See Rule 8 of Section 13.4 for one such example.)

2. Retain master copies and backups of all file elements (except scratch files) for the duration of the project. Files may be updated, or output on tape, but not deleted except as provided for in the backup policy. At the end of the project, the entire contents of project files may be dumped (to tape or printout) for the archives, and removed from the computer.

3. Retain a copy of all file and documentation updates in the archives. If an update was initiated by an engineering change order, enter appropriate cross-references into the project change-control log.

4. Maintain a cumulative record of the computer resources used to test the program for later evaluation of team productivity.

Measure CPU time, number of runs, etc., and enter these figures into the Project Notebook. The total CPU time per 1000 source card images is one such figure that can characterize the project's usage of developmental support resources.

14.8 DIAGNOSTIC PROCEDURES

In this section, I do not want to give a full set of diagnostic techniques, but, rather, how the tester should respond when errors are detected.

1. Do not discard test data, test code, dummy stubs, etc., when errors are found in the program. Rather, retain these in order to retest the program when the discrepancy is removed.

2. Submit Discrepancy Report forms into the archives for each failure detected, and summarize each such difficulty in the Software Test Report.

3. If a real-time program fails in consistency (non-repeatable errors), first, seek ways to make the program at least consistent, and then correct it.

4. Make all modifications to correct errors using standard software maintenance facilities for source and object text. No modifications (especially binary corrections) should be made while conducting a specific test. If a binary or other such modifications are necessary, then rerun the tests after the source program has been updated.

This discipline encourages more reliance upon testing as a program verification aid, rather than as a design-patching tool.

5. Determine the set of conditions that result in the error. Then submit the program to these conditions and monitor the execution as follows:

- a. If there are concurrent processes, monitor the interprocess communications. Typically, this requires examination of synchronization controls and significant parts of messages transmitted between processes. Such an examination indicates the flow of control under which the error occurs.
- b. Establish the relevant types of trace information for the processes of interest and trace the flow of control (and timing) within these processes to isolate the problem into narrowing subsets of candidates.
- c. Once the offending segment of code is isolated, examine the results of detailed computations (register contents and memory changes, if necessary) to isolate the specific cause.

6. Use simulation of hardware in concurrent program malfunctions to examine such things as:

- a. Whether a supposed process was executed or not.
- b. Whether test results up to a given point are correct.
- c. The sources of possibly erroneous data.
- d. Which of two or more events occurred first.

7. Seek abbreviated or scaled-down tests that are sufficient to invoke the malfunction as a measure toward reducing costs and hardships of computer usage.

8. If an error seems to appear because data is being destroyed, then check the scoping, either by tests or visual inspection, of the offending variables or files to find out where these items are being misused.

14.9 SUMMARY

This chapter has addressed disciplines that can be applied to an evolving program in order to increase its probable correctness upon completion. The disciplines border on formal Quality Assurance measures, which are the subject of the next chapter. The treatment of "Standard QA" to follow, in fact, has this discipline imbedded within it, but extends the concept of "correctness" and "testing" beyond the program top-down evolution to the more encompassing concepts that are needed to yield a sound, reliable, and well-documented piece of software.

XV. QUALITY ASSURANCE STANDARDS

In this chapter I shall discuss disciplines in support of software development which contribute to the quality of the delivered product, both functional and documentational, over and above the normal precautions and practices for good design, coding, development testing, and documentation that have been addressed in previous chapters. The disciplines in this category I call "Standard Quality Assurance," or, for short, just "QA."

Quality Assurance takes perhaps as many different definitions and roles in software production as there are producers of quality software. The general purpose of QA measures, however, is to minimize production problems by better planning and exercising better and tighter controls during the development of the product. The agency that is chartered to provide the required control functions is Quality Assurance, and many developers relegate QA measures to an organization separate from the development team in order to obtain unbiased, dispassionate confirmation that the product has met its requirements for deliverability. Many customers will insist on inspections by their own personnel.

However, I do not want to orient this chapter's approach to QA along organizational lines. I do not want to picture QA as a task separate from development, nor do I want to imply that it can be wholly accomplished by an agency integrated into the development team. Rather, I would like to set forth functions, disciplines, procedures, and philosophies for QA that can be independent of the organizational divisions of labor.

The envisioned situation is this: A program and its documentation have been (or are in the process of being) produced. At eventual points in the production, the developers have convinced *themselves* that a software unit is ready for configuration-controlled status, perhaps with liens toward future capability not yet operational, but coming. At this point, the customer, user, or operational organization seeks assurances that the software configuration satisfies delivery criteria.

The role I cast for Standard Quality Assurance is to provide this certification. The role of this chapter is to make this certification trustworthy.

15.1 STANDARD QA ACTIVITIES

Quality Assurance needs to be both effective and economical, in the sense that the expenditure of effort and costs to certify the product represent a justifiable cost savings in the software package life cycle. Ideally, duplication of effort among personnel should not be necessary. Whenever there is an area of interest both to the developers and to QA, the QA tasks should be in direct support of development and not in competition. For this reason, many QA measures can be integrated into the processes of design, documentation, coding, testing, configuration control, discrepancy reporting, and change control. In later verification and certification of the entire package, the developers should be in direct support of QA, and not in competition.

Fortunately, the concurrent design, coding, development testing, and documentation disciplines, made feasible by hierarchic, modular, structured programming, foster concurrent QA, as well. After all, the whole method so far described is predicated upon achieving a high degree of initial program correctness. Many of the rules given in previous chapters are QA measures. Therefore, much of the QA function is already integrated into the development process.

For example, part of the development cycle includes peer corroboration of design, coding, and developmental testing as a means for improving software quality (in addition to producing reliable software more quickly).

Part of the team organizational discipline is based on the presence of a Software Development Library and Project Archives to provide stable communications media and developmental support to team members. Audits necessary for QA also have access to these media.

Besides the informal QA measures within a project, however, there needs to be a formal, end-to-end demonstration of the software quality and a complete audit of the entire software package--the executable program and its documentation. These should check conformance to standards, design vs. functional specifications, code vs. design, test results vs. test specifications, performance vs. requirements, etc. Final certification is the verification seal for delivery.

The areas within which QA can function are [12]:

- a. Participation in design reviews.
- b. Review of documentation, listings, etc.
- c. Standards enforcement.
- d. Configuration control.
- e. Discrepancy reporting.
- f. Change control.
- g. Testing and test review.

An auxiliary activity [13] falling into the QA area is the gathering and dissemination of statistical data relating to coding, testing, later maintenance, etc., with respect to reliability and performance measures collected over many projects. Such data increases the understanding of the mechanisms by which software and software projects fail, calibrates the amounts and types of testing needed to achieve a given reliability, and monitors the differential utility of testing and QA measures as programs increase in reliability.

15.2 QA MEASURES DURING PROGRAM DEVELOPMENT

Although usually not made a formal part of the certification role of QA, perhaps one of the greatest contributors to software quality is the use of peer corroboration during each phase of the program development. Such corroboration is a QA mechanism because it promotes communication within the design group, tends to create and enforce standards, and encourages proper documentation and records.

Moreover, it is just plain, everyday, good engineering practice. As in all good engineering practices, a design should be *verified*. Design verification, as I mean it here, is a careful examination of the design by someone skilled in design, other than the designer himself. Perhaps the best choice for this job is the designer's supervisor; at the least, it should be a senior colleague. The purpose is to get a concurrence that the design at the current level is correct (i.e., that it will do what it is supposed to do) and is "good" by whatever criteria have been established for the project.

Another contributor to quality software is the availability of malfunction statistics within a project and across many similar projects. These statistics, plus a good set of standards, can drastically reduce the "learning curve" by indicating where projects typically get into trouble. Then, extra care, contingency planning, and similar measures can be applied as the cases warrant.

The availability of such statistics means that records need to be kept of the numbers and kinds of difficulties encountered, the time-distribution of such occurrences (to show when preventive measures will likely be needed), and the seriousness of faults. The Project Notebook is an ideal location for such statistics during the project development phases; a summary report at the end of development can then be incorporated into organization (or industry) statistics.

I have already given rules which include these QA measures in previous chapters, so they warrant no repetition here.

15.3 SOFTWARE TESTING CHARACTERISTICS

Dijkstra's remark at the 1969 NATO Software Engineering conference [14] to the effect that "testing only proves the existence, rather than the absence, of bugs" is a widely quoted truth about the nature of a necessary process in software production. Frankly, however, the prospects for program reliability are not as bleak as that statement may make them seem. For one thing, we have seen in Chapter 9 that our confidence in statements of the form "there are no more errors in the program" can be raised to any desired level by enough testing. The works of Wolverton and Schick [15], Musa [16], and Jelinski and Moranda [17], which I will say more about here, reveal much about the mean times between discoveries of software anomalies via testing (and usage).

It is important for programmers and managers alike to understand just what it is that testing does and does not do, how much it costs, how long it takes, and what things can and cannot be inferred from test results.

Almost every software project seems to enter a phase where it is "90% complete," in which it appears to remain for a very disproportionate length of time. Much of this time, as it turns out, is spent discovering and repairing anomalies in the program, operations manuals, or program requirements. The numbers and kinds of anomalies in a software package are matters of *fact* and not matters of probability; however, since only a relatively small portion of a large program's documentation and response can ever be verified in a practical sense, the process of discovering anomalies appears to be a random process.

Repairing an anomaly requires study, software alteration, and, then, reverification. Because the differing kinds of anomalies exhibit a range of difficulty, and because human interaction is always required, the repair rate also appears to be a random quantity.

Statistical models of discovery and repair provide us with a basis for extrapolating past performance characteristics into the future, as a means of predicting when testing will be complete, or what the expected reliability by a certain date will be. Such information, if gained early enough in a project, can point out likely problem areas and permit the reallocation of resources as necessary to realign completion dates with committed capabilities.

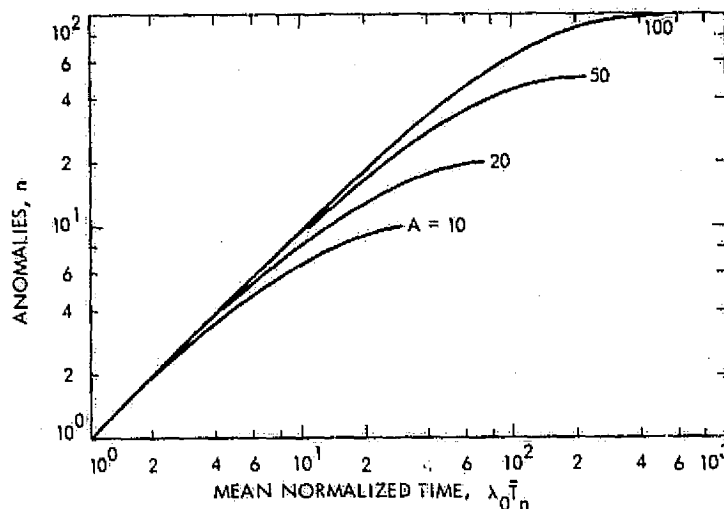
15.3.1 A Random Discovery Model

The simple model of anomaly discovery used in Chapter 9 (viz., the probability that an untried test will find a new anomaly is assumed proportional to the number of anomalies yet remaining) seems to apply to many kinds of testing, especially in large systems with only a relatively few, independent errors. This model predicts that an ensemble of identical projects will find anomalies at a certain average rate, and that there will be deviations about this mean of a certain magnitude. The average amount of effort, here represented in units of the time applied, T_n , required to detect n anomalies is given by the formula

$$\bar{T}_n = \text{avg}(T_n) = \frac{1}{\beta} \sum_{k=0}^{n-1} \frac{1}{A - k}$$

The behavior of this average is plotted in Figure 15-1. The mean-square variance about this average time is similar in form,

$$\text{var}(T_n) = \frac{1}{\beta^2} \sum_{k=0}^{n-1} \frac{1}{(A - k)^2}$$

Figure 15-1. Normalized mean time to reach n th anomaly

This latter function is not plotted, but the normalized ratio, variance to squared-mean is graphed in Figure 15-2. Both of these figures display dependencies upon n and the true number of anomalies, A , which, of course, is unknown. The first graph is, additionally, a function of a testing productivity factor β (it cancels out in the ratio). The discovery-rate graph is plotted for several assumed values of A , but constant initial discovery rate, $\lambda_0 = \beta A$.

The ratio, $\text{avg}(T_n)/\text{avg}(T_A)$, represents the average relative completion status during anomaly discovery. The behavior of this ratio is shown in Figure 15-3. It shows, for example, that if there are 100 anomalies, then when 90% has been found, only about 44% of the total required effort has been expended. It costs 56% of the testing effort to find the last 10% of the bugs! And this cost increases to about 70% if there are 1000 anomalies. A clear reason why software should have as high a reliability as possible prior to acceptance tests!

Discovering anomalies tells us two things: n , the number found so far, and the efforts expended to discover each, denoted t_k for $k = 1, \dots, n$. The maximum-likelihood estimations of A and β are values simultaneously satisfying the formulas:

$$AT_n - \frac{n}{\beta} = \sum_{k=1}^n (k-1)t_k$$

$$\beta T_n = \sum_{k=0}^{n-1} \frac{1}{A-k}$$

The second of these equations substituted into the first renders the estimation of A independent of β . Figure 15-4 shows this relationship, with the estimated value of A on the abscissa versus a calculation based on measured values on the ordinate. To find the estimated A and β , one merely computes the indicated ratio, locates the proper n -curve, reads the corresponding A , and substitutes back into the last equation, above, for β . Note that small variations in the measurements may yield wild variations in the predictions unless n is an appreciable fraction of A .

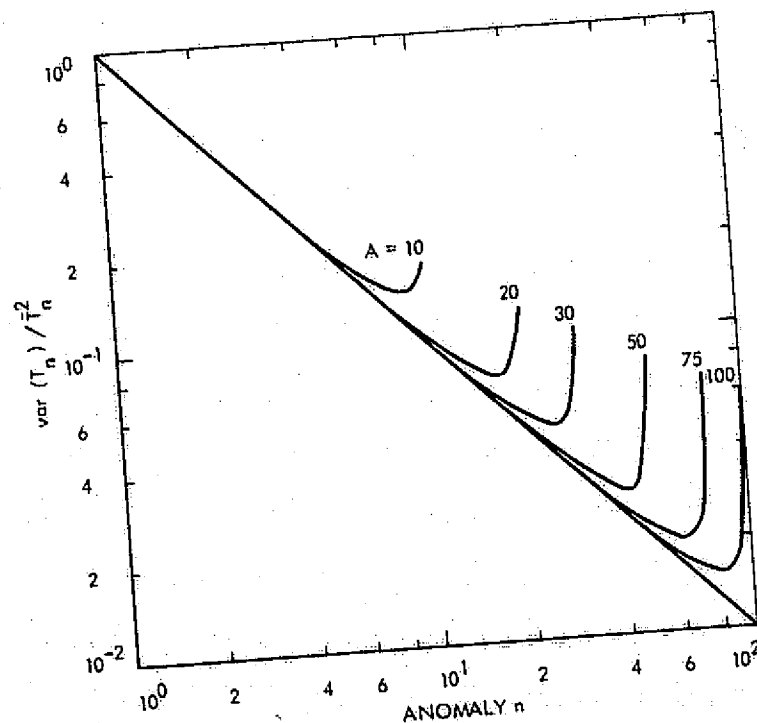


Figure 15-2. Normalized variance in time to detect n th anomaly

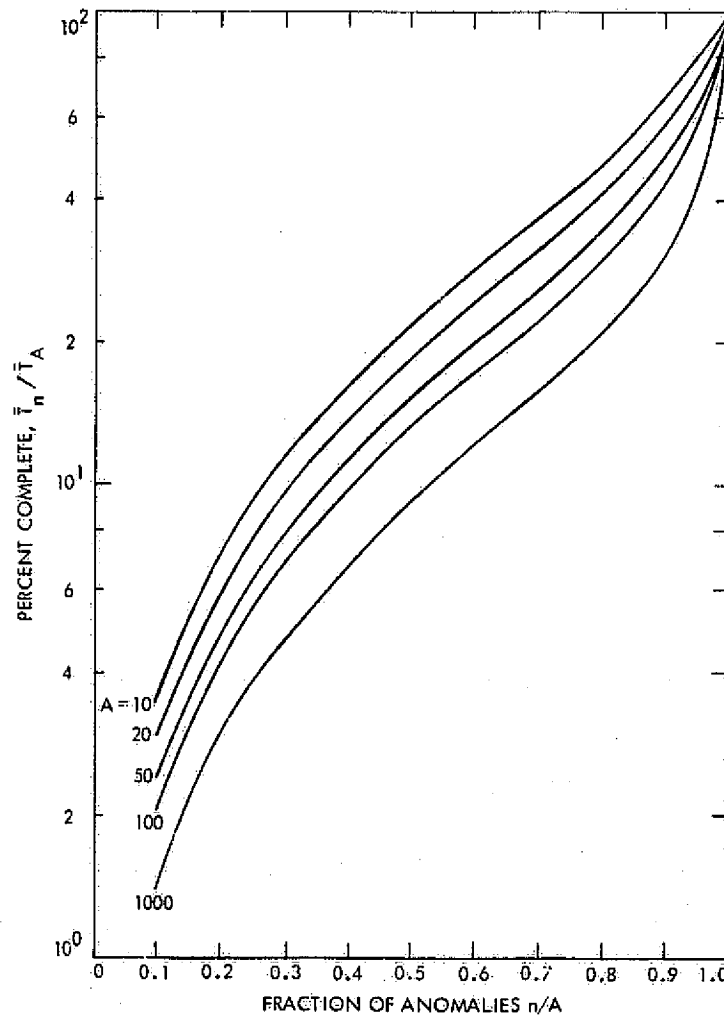


Figure 15-3. Normalized expected completion ratio vs fraction of anomalies discovered

As a result of this model, there are several important conclusions that one may draw about anomaly discovery. The information contained in the four figures presented so far clearly points out the following facts:

- a. After detecting n errors, one may estimate the total number of errors present in the program. Estimation accuracy is higher in programs with larger numbers of errors to begin with, and then only after a significant fraction of them have already been found.
- b. Having estimated that a significant fraction, say, 90%, of the anomalies has been found, one may estimate (Figure 15-3) how much

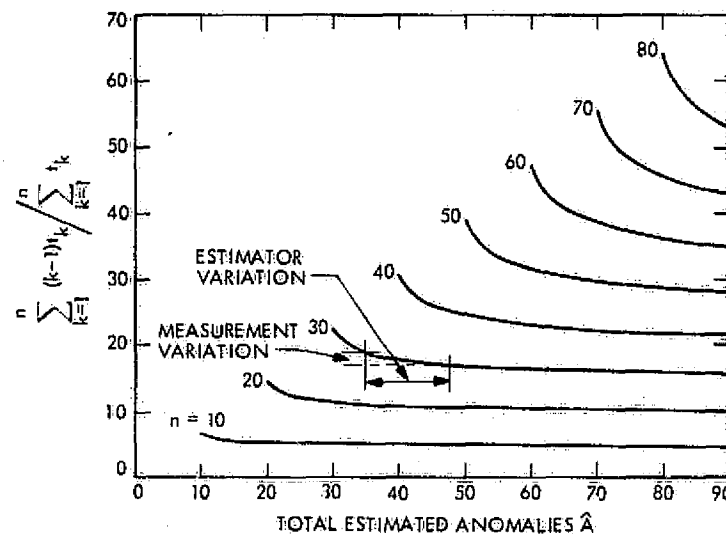


Figure 15-4. Measurement ratio vs maximum likelihood estimator for number of anomalies, given the number n of anomalies discovered so far

effort is yet required to find the remaining 10%. This remaining 10% will almost certainly require a disproportionate amount of effort to discover. However, that effort requirement is not unreasonable or unnatural. It is one of the laws of software.

- c. The variation in the total effort that an ensemble of identical projects would require to discover n anomalies increases with n . That is not unusual; most random phenomena display increased variations when the mean value also increases. However, percentage variations usually decrease as the size of the sample space increases. The anomaly discovery process, too, displays this normal behavior (Figure 15-2) up until about 80-85% of the anomalies have been found. Then, the effort required to find the final 15-20% of the anomalies can be expected to deviate from the expected time by wider and wider margins, even on a percentage effort basis. If schedules are being set or forecasts being made requiring contingencies or reapportionment of resources, such variances need to be taken into account in order to avert schedule disasters.
- d. The above behavior supposes that a constant level of effort is being applied toward finding anomalies using methods which find these errors as a purely random event. Therefore, these discovery characteristics will most certainly be in effect unless some testing methods can be brought to bear which either intensifies the effort or else organizes the test cases in such a way that errors are more

deterministically found. (The tests in this and the preceding chapter are directed specifically toward this latter end.)

15.3.2 An Anomaly Repair Model

Before addressing a more general find-and-fix model for anomaly performance, let me first presume that a number of anomalies have been discovered, that a constant effort (again measured in units of time) is being applied toward their repair, and that the probability that the anomaly currently being worked on can be fixed by applying only an additional Δt effort is proportional to that effort, viz., $\mu \Delta t$ for some constant effort factor μ . Such presumptions about the repair process lead to simple calculations for the statistics of the required repair effort, T_m , for m anomalies:

$$\bar{T}_m = \text{avg}(T_m) = \frac{m}{\mu}$$

$$\text{var}(T_m) = \frac{m}{\mu^2}$$

$$\frac{\text{var}(T_m)}{\bar{T}_m^2} = \frac{1}{m}$$

This model predicts that all of the n known anomalies will be repaired uniformly in time with growing absolute, but decreasing relative, uncertainty. Each anomaly requires an average time $1/\mu$ to repair, and the standard deviation from this average is also $1/\mu$.

This model also infers that once an anomaly is discovered its repair is a fairly well-ordered process. The repair rate statistics do not depend on the total number found so far, for example. The characteristics of this model, in fact, fit fairly well with observable statistics in actual projects. A best-fit line through observable data, then, is a measure of μ .

Whenever the initial find-rate, λ_0 in the preceding section, exceeds μ , the constant fix-rate, there will be a certain span of time during which there will always be open anomalies: detected, but, as yet, unrepaired. However, there is an eventual time at which the decreasing discovery rate falls off to the point at which all known anomalies will have been removed. When this condition (called "zero defects") occurs, the repair rate, of course, drops to zero, and the model no longer applies.

The average time to zero defects can be estimated from the discovery and repair models and data by extrapolation (Figure 15-5). This will occur roughly where the two curves cross, a condition given by

$$\frac{n\lambda_0}{\mu} = A \sum_{k=0}^{n-1} \frac{1}{A-k}$$

On reaching zero defects, there remain $A - n$ anomalies yet to be found. As it turns out, the ratio $r = 1 - (n/A)$, or the fraction yet undiscovered at the time of first zero defects, is a function principally of the initial rates of progress, as shown in Figure 15-6. According to this figure, an initial find-fix ratio in excess of about 2.2 will result in fewer than 10% lurking anomalies after zero defects have been reached.

15.3.3 Effects of Variation in Effort

The anomaly models in the previous section were based on constant levels of effort being applied to finding and fixing of anomalies. The assumption of constant effort is tantamount to equating time and cumulative expended effort in the equations and figures presented. In actuality, however, the effort profile may be variable for many reasons, among which are manpower phasing, availability of computer resources, and availability of software resources. Thus, the accumulation rate of applied effort may differ greatly from the linearly rising accumulation of time.

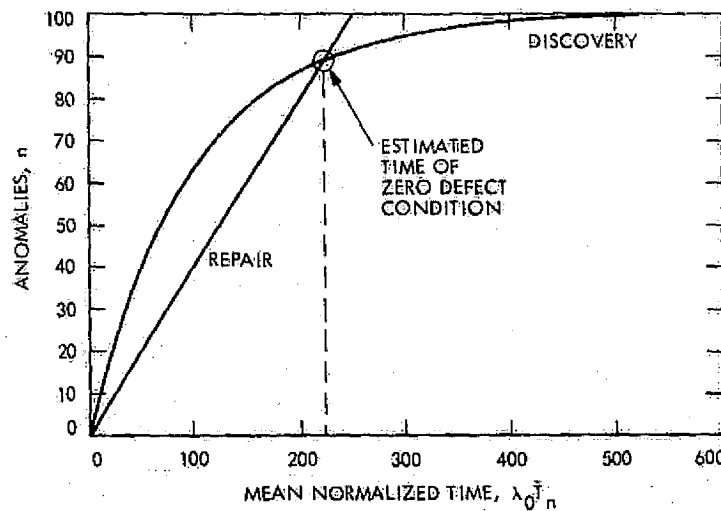


Figure 15-5. Plot of normalized anomaly discovery time for $A = 100$ and normalized anomaly repair time for $\lambda_0/\mu = 2.5$

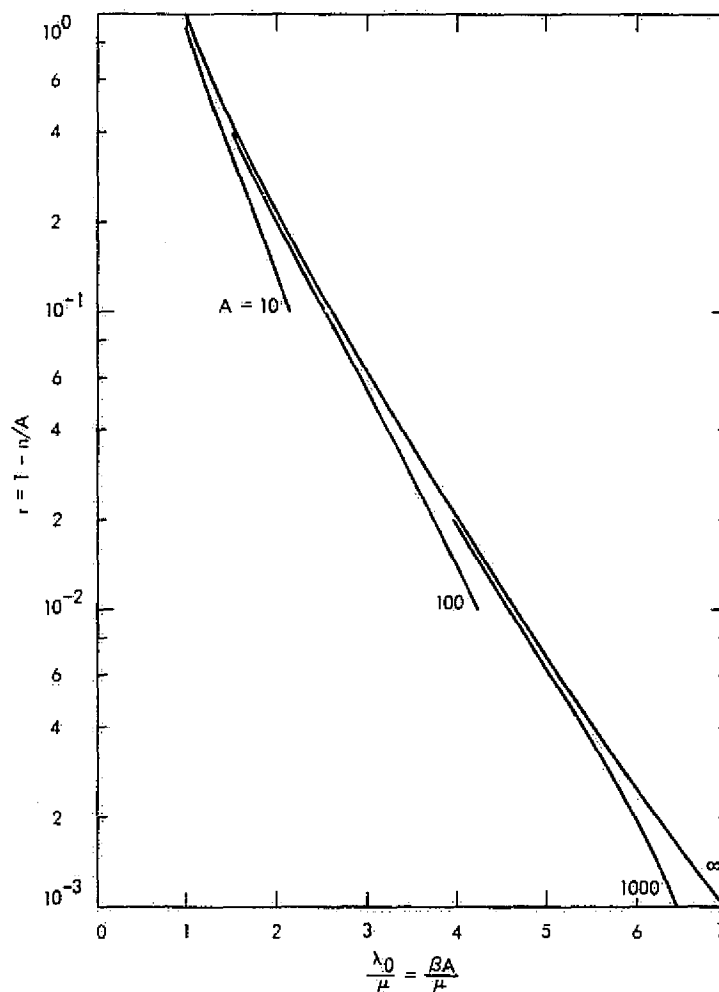


Figure 15-6. Mean time intersection approximation to fraction of anomalies remaining at time of first zero-defect condition

It is typical that effort during the early testing is at a rather lower level than later on, because "things are getting up to speed." Effort is being put into planning, coordination, training, and resource acquisition rather than into actual testing. Toward the end, effort may drop again, to the level supported by operations and sustaining personnel. This phenomenon is illustrated in Figure 15-7; in this figure, 1 person is applied for 7 days, 7 for 25, 4 for 22, and 2 for 30 days.

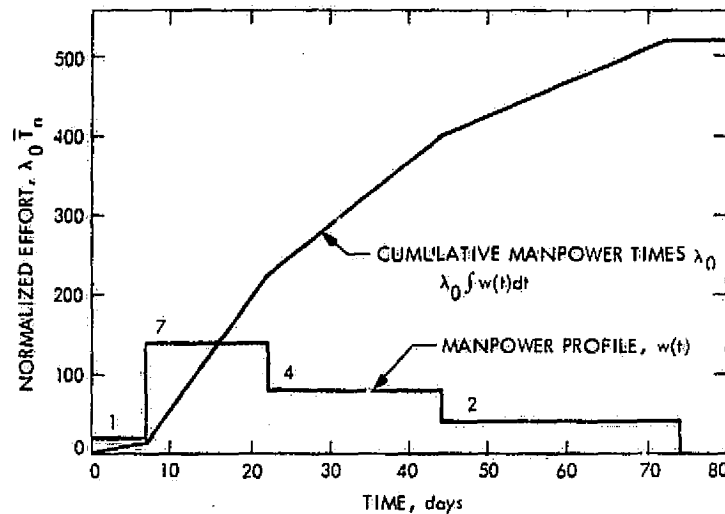


Figure 15-7. Work-level profile and cumulative effort normalized by initial discovery rate of $\lambda_0 = 2$ anomalies per man-day

Compensation for varying levels of effort (at the same productivity level) is accomplished by replacing the time variable in figures and formulas by the accumulated hours of work up to that point. An illustration of this principle appears in Figure 15-8.

Conversely, one may compensate in reverse: That is, the anomaly-versus-effort behavior may be plotted and analyzed using the previous estimators, then translated via the projected work-level profile into estimates of anomaly status at future dates.

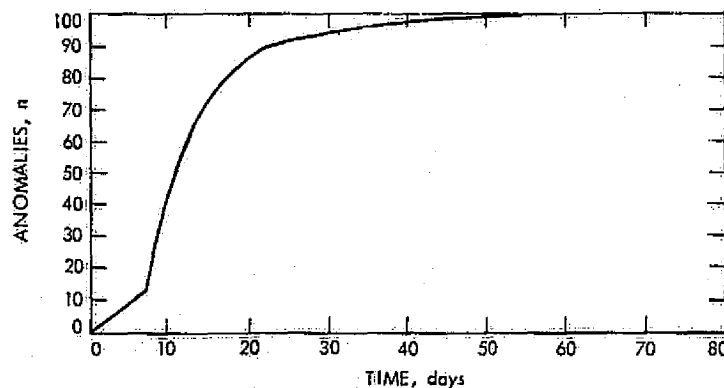


Figure 15-8. Effect of work-level profile on discovery of anomalies for Figure 15-7 and $\lambda_0 = 2$ anomalies per man-day

When work levels change, the productivity factors are also apt to change merely because a different set of individuals are performing the tasks. If work levels remain constant for long enough periods of time, the relative productivity per individual can be estimated and also factored into calculations. An example of such re-evaluations of productivity is given in the next section.

15.3.4 Cascaded Testing

Even when work profile effects are factored in, it is frequently the case that discovery of anomalies takes place in various environments for supposed economic reasons. Certain anomalies may, therefore, not be discoverable in one environment but, perhaps, discoverable in another, due to different software or hardware configurations.

For example, if a set of real-time programs are first tested outside the real-time environment, those anomalies that are due to the real-time interaction among processes are, perhaps, not discoverable by any means until the programs are integrated into their true operational environment.

In such cases, only a lesser number, say, A_1 , of the total anomalies will be findable during the first phase of testing, no matter how long carried forth. If the second phase takes place in the final operations environment and begins after n anomalies in the first environment are found, then $A - n$ become discoverable during this stage.

This phenomenon is illustrated in Figure 15-9, which assumes a constant level of effort β . $A = 100$ total anomalies, and $A_1 = 50$ findable during the first phase. Switching to the operational environment takes place at 45 errors, or 90% of those that can be found in that environment. As may be seen, multi-stage testing may take a significantly longer time (31% in the illustrated case) to find all anomalies.

Moreover, the test time requirements on usage of the operational facility are about the same (52 vs. 46 days) in either case. Thus, an expenditure of 16 days saved 6 days in the operational facility. Actual dollar costs for such situations need to be determined in order to justify multi-stage testing when it can be avoided. Unless there are other overriding constraints that mandate multi-stage testing, this form of "bottom-up" anomaly discovery plan is likely not to be cost- and schedule-effective.

Figure 15-10 is an actual anomaly history; unfortunately, the work profile and productivity information was not available so that a more detailed comparison with the theoretical models was not possible. However, the reader may note that all of the predicted elements are present: The effects

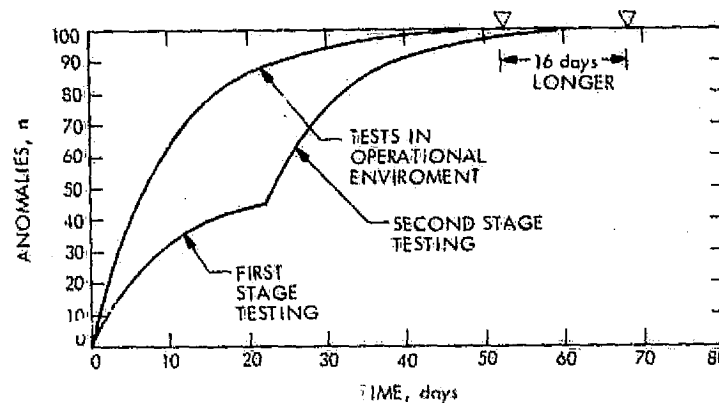


Figure 15-9. Cascaded discovery of 100 anomalies in 2 stages where 50 anomalies only were visible during first stage for same $\beta = 0.1$

of low-level effort during the start of testing, the decreasing rate of anomaly discovery in the sub-operational environment, the increasing rate thereafter in the operational environment, and the ultimate leveling off as testing continued. The figure indicates the relative levels of efforts needed to make the plotted curve best fit the measured points (crosses). The estimated final number of anomalies was about 306, but only about 218 were findable prior to operational transfer.

15.4 RULES FOR ACCEPTANCE TESTING AND CERTIFICATION

The ultimate goal of a software package is that it operates error-free and meets operational requirements. The proof that it performs as it should can be accomplished only by adequate demonstration and stringent testing. The following rules do not address organizational but functional responsibilities.

1. Determine the type and extent of deliverable items and testing needed to satisfy the acceptance requirements.
2. Develop and document test plans and procedures that define what to test, how and when it is to be tested, what to look for during and after the running of the tests, and what to do with materials supplied for testing. Descriptions of tests in the Test Specification in the SSD should include [12]:
 - a. Test objectives and procedures to achieve those objectives.

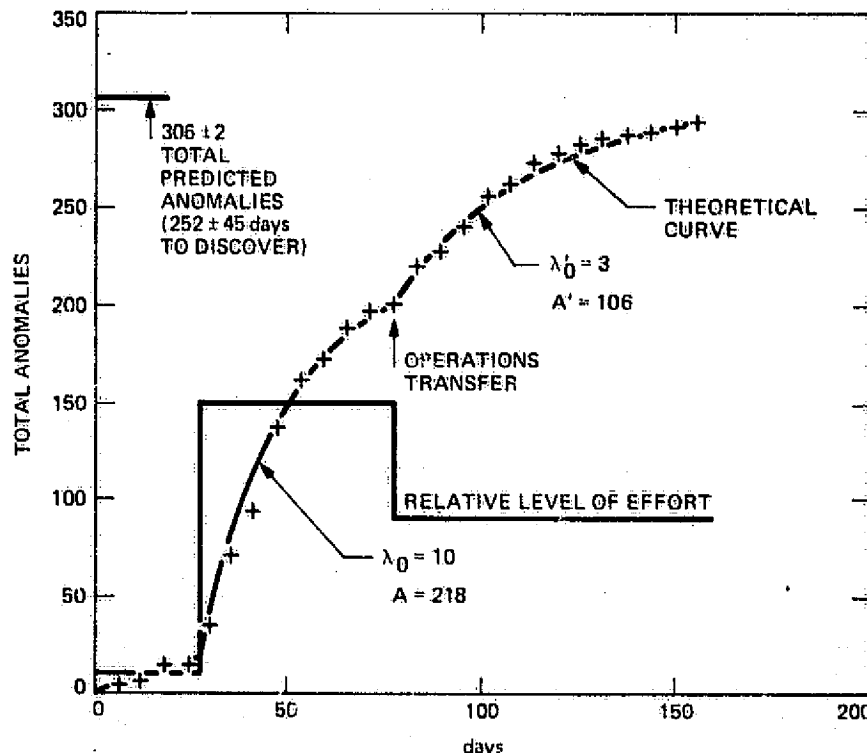


Figure 15-10. Application of theoretical anomaly discovery model to actual project involving about 100,000 lines of assembly language code

- b. Equipment (hardware) required to support and complete the test.
- c. Software required to support the test.
- d. Limitations relative to the accomplishment of the test, such as timing, hardware/software interaction, etc.
- e. Methods to be used to verify performance of the software, such as inspection, review of analytical data, visual displays, and analysis of output.
- f. Acceptance criteria in terms of presence or absence of specified characteristics, such as inputs, outputs, limits, ranges of input/output, amount of data, and critical values.
- g. Test sequence or list of ordered steps to be executed to perform the test.
- h. Initiation instructions relative to bringing up the system into the test configuration.

i. Termination and restart contingency instructions.

3. Schedule and perform tests, complete and end-to-end, in accordance with the test plan, but first as "validation tests" or "rehearsals" that are basically gross checks of the coding against functional specifications and validations of the test procedure documentation. Encourage the development-testing personnel to perform these validations themselves, to attend the rehearsals, or to monitor and critique the testing results. If anomalies noted during these dry runs uncover program discrepancies, advise the project manager for appropriate action.

The SSD "as built" or acceptance readiness review freezes the configuration for delivery tests. All changes subsequent to the freeze require formal anomaly reporting and archiving of red-lined items. An anomaly summary should be presented at the acceptance review.

4. Ascertain that the program response meets its functional requirements based on tests derived primarily from the user/operator manual(s). If timing is not critical, enable flowline-trace features to gauge what percentage of the actual instruction code has been tested during this functional testing.

5. Ascertain that the program response meets performance requirements (speed, memory usage, etc.) by benchmark tests derived primarily in response to system and environmental requirements, but using tests developed for functional testing. That is, certify the performance parameters on the same portion of the program certified functionally.

6. Perform final acceptance tests only in the actual operational environment or in an exact duplicate of the actual. Otherwise, attach a QA lien to the certification for later removal, after delivery and contingent upon stated retest criteria.

Simulation in other than the actual operational environment is useful as precursory testing or perhaps as final development testing; however, the testing on which acceptance is based should be performed as stated. The acceptance test procedures should include the generation and loading of the program to be tested from the delivery source tapes/disks.

7. Prescribe tests that, if successful, yield a definable level of confidence in program reliability.

Test all modes of operation, all logical conditions, and all data conditions using confidence-level techniques when

they can be applied. See Section 9.4 for one such technique.

8. Subject the program to anomalous data (containing errors, out-of-range values, wrong data sets, etc.), overload conditions, and improper operations interfaces (such as control-sequence errors) to verify proper program recovery in all cases.

9. If errors are found during certification that are due to the user/operator manuals, rather than the program, arrange to have the manuals corrected before delivery, or else state such errors as a lien against certification, later to be removed.

10. Record all deviations from test procedures, impound all test output, and recover all test materials (that are not part of the deliverable configuration).

11. Classify all anomalies by priority or seriousness in standard categories, such as:

- a. Removal of anomaly is critical for usage of the software as required.
- b. Anomaly degrades performance or increases operational risks.
- c. Anomaly does not prevent software from being used successfully, but it is undesirable in that it requires user/operator reorientation or work-around.

15.5 SOFTWARE AUDITS

In general, a QA Software Audit [13] consists of a visual inspection of documents to determine if they meet certain known standards and requirements. An audit is not intended to review the conceptual approach to a solution of a problem or to a design. It should simply provide definite assurance that certain documents are in accordance with what other documents have prescribed. When instances of non-conformance are found, an audit report should explicitly detail the location and type of non-conformance.

If QA audits are carried concurrently with design, coding, documentation, etc., such audits decrease the possibility of oversights, avert misconceptions that could result in major rework or liens, prevent the augmentation or alteration of the design unilaterally at later design stages without proper approvals, and encourage a uniform, standardized design.

Let me stress that *validating* the module algorithm is *not* a part of the *audit*. Rather, the auditing process is purely a "bookkeeping" job, something which keeps the design "honest." It can, and preferably should, be done by someone other than the producer himself, as I stated earlier. If performed by a peer, then the peer can both audit and validate.

15.5.1 General Rules for QA Audits

1. Prior to the performance of an audit, make a checklist identifying the practices to be used, the documentation segments to be present, the standards to be met, and the problems to be avoided.
2. Identify the limits of the audit in the current development phase, such as which modules are being given what type of audit (design vs. requirements, code vs. flowcharts, etc.).
3. Obtain, for auditing purposes, one or more copies of the documentation that represent the product at the current audit phase.
4. Utilize personnel for QA who have a good general understanding of the project in particular and a sound practical experience in QA techniques.
5. Generate a formal QA report to the cognizant project manager for his information and action.
6. Do not transfer software from development to operational status unless a QA audit has been made and certifies that discrepancies found have been properly resolved (perhaps by the attachment of a lien to the certification stating temporary waivers to certain discrepancies).
7. Be watchful for omissions in content or apparent contradictions that may become sources of confusion in later work.
8. Flag items as discrepant where clarity or exactness seems to be needed, but is not provided.
9. Check the format of each document against its prescribed outline for conformity, as well as to catch omitted segments.
10. Verify that stated program interface specifications under audit match the cited environmental configuration descriptions being assumed.
11. Verify the existence of all required documents, that these have been completed to their specified levels, and that the quality of each is acceptable.

15.5.2 Rules for Auditing Software Specifications

1. Affirm that every function in the Software Functional Specification is traceable to a functional requirement, and, conversely, that every functional requirement has been responded to in the SFS.

2. Affirm that every function in the SFS appears either as a module or is imbedded identifiably within some module in the Programming Specification portion of the SSD.

3. Audit flowcharts, narratives, mode diagrams, data-flow diagrams, and all other forms of design specification against structural and documentation standards, such as those given in Chapter 12. Specifically, make sure that within procedural specifications:

- a. All decisions in each module test explicit, determinable condition flags either defined within that module, passed to it as an argument (or globally), or returned to it by one of its submodules.
- b. Each data structure referenced within a module appears in the Glossary and in a Data Structure Definition Table (if its description is incomplete in the Glossary).
- c. All submodules of a given module perform actions identifiable as subfunctions of the stated function of the given module. Flag as discrepant those subfunctions that seem to be missing or extra.
- d. All horizontally striped modules have corresponding flowcharts (or equivalent) at the next design tier, properly cross-referenced by Dewey-decimal notation.
- e. All vertically striped modules have corresponding interface descriptions in an appropriate External Module Interface Description.
- f. All flowchart boxes (or equivalent) which call either external or internal subroutines have their Dewey-decimal number inserted in the subroutine cross-reference table.
- g. Flowcharted specifications are drawn and annotated as per Section 12.7.2. There must be accompanying narrative, documented as per Section 12.7.3, for each flowchart. Alternative procedural specifications, such as CRISP-PDL, should be documented as per Section 12.7.5.

4. Keep a record of each flowchart (or equivalent) audited by name, Dewey-decimal identifier and version date. Assign discrepancies to categories, and give the number of discrepancies found in each category.

5. Affirm that real-time programs have documented standards to provide consistency in concurrent programs. Specifically, verify that methods are addressed which treat:

- a. Arbitration of shared resources.
- b. Anti-deadlock measures.
- c. Thrashing.
- d. Recovery from deadline failure.

15.5.3 Rules for Code Auditing

1. Verify that any programming constraints or requirements set forth in the SRD, SDD, or SSD are being met, such as:

- a. Programming language(s) used.
- b. Use of existing capabilities.
- c. Compile-unit modularization.
- d. Reentrancy considerations.

2. Obtain and use as an audit guide any project-peculiar standards, such as:

- a. Register names, usage standards.
- b. Module-to-submodule linking methods.
- c. Special handling of certain design specifications.
- d. Definition of compiler parameters, literals, internal program labels, and special storage structures.

3. Affirm that the code "matches" the design. In particular, verify that:

- a. The code takes the same modular form as the design, except as provided for in special programming standards or waivers.
- b. No functions have been omitted, nor have any extra functions been inserted, except as necessary coding considerations to support the given design.
- c. Coded modules are properly cross-referenced to the design and annotated so as to make clear that the code does in fact match the design, box for box, on the flowchart (or equivalent).
- d. There are no compiler diagnostics or errors.

4. Inspect the explanatory documentation provided in support of bridging the design to the code, such as:

- a. Cross-reference lists (charts, or equivalent, compile modules/files).

- b. Index-register usage tables or standards.
 - c. Glossary of special variables or literals not in the design and not easily defined by name or use context.
 - d. Memory use map.
 - e. Timing diagrams.
 - f. Interrupt-handling procedures and relationships.
 - g. File, table, and data set descriptions.
 - h. Examples of input and associated output.
 - i. Listing of special flags, pointers and other indicators together with their usage (which routines, areas or times of applicability).
 - j. Commentary describing features of code that link performance to design documents.
 - k. Lists of error conditions, codes, and messages, cross-referenced to both the design charts and the code itself.
 - l. Restrictions on the use of code that is particularly sensitive to changes in design (mainly time and memory space, but also functional limits to subroutines, etc.).
 - m. Data usage, such as shared public files vs. restricted-access files.
 - n. Hardware or software constraints.
 - o. Use of privileged instructions.
5. Check that measures specified to provide consistency in real-time and concurrent programs are actually employed on shared resources in the prescribed way.
6. Enter QA approval initials and date into the module headers of all approved code modules.

15.5.4 Rules for Development Test Audits

Having QA active during the module development testing is a measure taken to shorten the time required for acceptance testing and certification. For example, if QA certifiers can see that certain testing required for acceptance has been performed as a routine correctness measure during development, then such tests may not need to be rerun for certification.

1. Ascertain whether a given development test specification fulfills an acceptance test criterion. Mark such tests and observe the results of those in partial fulfillment of certification requirements.

In order for a given development test specification to qualify as a partial acceptance test, the development test must conform exactly to the (partial) acceptance criteria: All functions will have been demonstrated as required, no new functions or effects will appear, and any failure that may mar the result cannot have been the result of a software anomaly.

2. Perform qualified development tests in partial fulfillment of acceptance tests using the exact same modules and data spaces as exist in the final product. If any such modules have been altered, or if data storage has been rearranged, the tests should be recertified.

3. Audit the test code and test data used in development tests that also apply to certification. Verify that the test code matches the test design, that the test data invokes the proper required modes of operation, and that the program response echoes the required response.

4. Audit the software test archives to verify that stated development tests have all been run, in the prescribed manner, and yield the reported results.

5. Summarize the results of such development-supportive QA efforts so as to show:

- a. Coverage of functional requirements verified.
- b. Error modes and recoveries demonstrated.
- c. Timing, as appropriate.
- d. Validity of results by cross-reference to archived tests or other documentation.
- e. Signatures of test conductors, observers, or acceptance personnel.
- f. Names, Dewey-decimal identifier, and version date of the certified partial-acceptance modules.

15.5.5 Rules for Acceptance Test Audits

1. Affirm that the program and its environment are under configuration control, and that the configuration being audited is that which will be (or is being, or has been) tested.

2. Verify that test procedures exist in accordance with Acceptance Requirements.

3. Identify those parts of the program that have had inadequate development testing, so that they may properly receive more intensive tests during certification. Similarly, identify functional parts of the program that

have high priority, high reliability, or high impact on operational requirements, so that they may also receive more intensive testing.

4. Verify that all required test materials, such as object program tapes, test data, etc., are certified by QA and are under configuration control before acceptance tests are performed.

5. Verify that all test equipment required to perform the tests are available and calibrated before acceptance tests are run.

6. Verify the proper conduct of the test, such as:

- a. The test conductor has brought the system up properly.
- b. All conditions of test procedures are observed.
- c. Each step in each test is executed in sequence without exception, using certified data supplied in the correct sequence.

15.6 DOCUMENTATION OF QA ACTIVITIES

Reporting and record keeping of QA activities is an administrative function that will assist the QA activity itself, as well as project management, in tracking the progress of a software development and evaluating its progress against milestones. Several kinds of reports are useful, among which are the configuration status, discrepancy status, and the change status.

Configuration status reports are the week-by-week summary reports of the progress, in terms of modules designed (and documented), accepted, coded, and checked out, primarily for management. It also can contain such information as the current version, last update date, file name or tape number, and statistics, such as core usage, etc. Figure 15-11 is a sample report of this type.

In larger projects, where the configuration evolves or changes noticeably from day to day, there may be the need for more frequent and detailed communication among the developers, in the form of an automated "Daily Software Overview." Such a report, for example, may inform its readers of new modules added or corrected, problems discovered or fixed, the file names of current and test versions of the evolving system, newly created test data, etc.

A discrepancy status report is another invaluable report, providing management at predetermined intervals with a summary of the numbers and types of problems being encountered, the rate of closure of the open

CONFIGURATION STATUS REPORT

PROJECT: MBASIC Implementation

Report Date: 6/13/75

NOTES: Performance relative to schedule established 2/1/75 based on information as of 6/1/75.

CHART CATEGORY	MODULE TOTAL	CODE START	DEV TEST	QA CHECK	SCHEDULE:	
					START	COMPLETE
1	1	1	1	1	02/01/75	03/01/75
2	3	3	3	3	03/01/75	04/01/75
3	5	1	1	1	07/15/75	08/15/75
4	182	165	148	5	04/01/75	07/15/75
5	12	0	0	0	07/01/75	08/01/75
6	183	45	1	0	04/01/75	10/15/75
7	1	45	1	0	03/15/75	04/01/75
8	1	1	1	1	03/15/75	04/01/75
9	1	1	1	1	04/01/75	04/15/75
P	122	3	0	0	06/07/75	09/20/75
R	48	0	0	0	08/20/75	11/15/75
I	8	0	0	0	10/01/75	11/01/75
U	96	7	3	6	07/15/75	03/15/76
T	8	1	1	1	10/15/75	11/20/75
E	166	0	0	0	08/10/75	05/10/76
TOTALS	837	229	161	20		

PERCENTAGE SUMMARY

CHART CATEGORY	CODE START	DEV TEST	QA CHECK
1	100.00 LA	100.00 LA	100.00 LA
2	100.00 LA	100.00 LA	100.00 LA
3	20.00 NS	20.00 NS	20.00 NS
4	90.66 OK	81.32 OK	2.75 NG
5	0.00 NS	0.00 NS	0.00 NS
6	24.59 SL	0.55 SL	0.00 SL
7	100.00 OK	100.00 OK	100.00 OK
8	100.00 OK	100.00 OK	100.00 OK
9	100.00 OK	100.00 OK	100.00 OK
P	2.46 NS	0.00 NS	0.00 NS
R	0.00 NS	0.00 NS	0.00 NS
I	0.00 NS	0.00 NS	0.00 NS
U	7.29 NS	3.13 NS	6.25 NS
T	12.50 NS	12.50 NS	12.50 NS
E	0.00 NS	0.00 NS	0.00 NS
TOTALS	27.36	19.24	2.39

OK On Schedule
 LA Done But Late
 NS Not Started (w.r.t. Schedule)
 SL Behind Schedule

Figure 15-11. A summary configuration status report of coding activity

items, and perhaps the level of effort in closing such discrepancies. Figure 15-12 shows a plot of a project discrepancy history that might accompany such a report.

When engaged in a large and complex software development, there will be many requests for changes to various modules during the program's evolutionary growth to completion. Since there can be no change to a controlled software item without first obtaining an approval for the change, it is imperative that the status of each change request be monitored until it has been completed or cancelled. The change status report is a summary report of such activity, provided at predetermined intervals to the project manager. Figure 15-13 illustrates the content of an engineering change log summary.

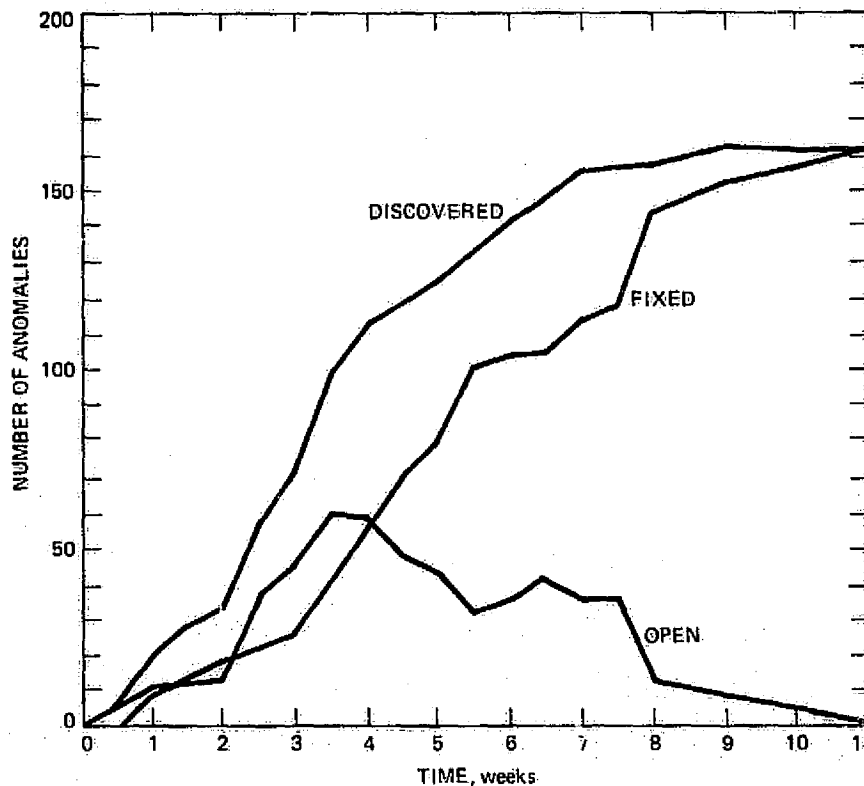


Figure 15-12. Anomaly report history for acceptance testing of 1973 Deep Space Network Telemetry/Command Processor software subsystem (program was real-time, non-structured, developed by "classical" methods)

Software Change Request						
Software Item:				Req. No.:		
				SW ID:		
				Date:		
Submitted By:		Phone:		Attachment: Yes <input type="checkbox"/> No <input type="checkbox"/>		
Version	Config/OS	Reply Date	Need Date	Anomaly No.	Cat-Prior	
Change Description						
Justification						
Other Items Affected						
Action	Disapproved	<input type="checkbox"/>	Grounds _____			
	Analysis recommended	<input type="checkbox"/>				
	Approved	<input type="checkbox"/>	Proviso _____			
Comments						
Signature		Date	Signature		Date	

Figure 15-14. Software Change Request form

specifications meet requirements, code matches specifications, and tests verify that the program runs without flaws.

This Software Test Report (STR) is identified in Figure 2-12 as the "Acceptance Certification" documentation. Figure 15-15 presents a candidate graphical outline of the STR, in which are shown summary reports of all testing and audit activities. The first section of the report is an abstract of the QA findings; the second addresses plans, resources, support, and applicable QA standards and conventions needed to read the remaining material. The remaining sections document the findings of testing and auditing the software to be delivered. A more detailed outline appears in Appendix J.

15.7 RULES FOR SECURITY, INTEGRITY, AND CONFIGURATION CONTROL

When software development items are produced, certified, and placed under project control, they must not again be altered or changed in any way without proper approvals, testing, documentation, and notification of concerned personnel. The Software Development Library is charged with making everything in the software development visible to all concerned, while keeping the lock on approved-status material. Such measures, since they deal with program integrity, qualify as QA functions. The following rules are guidelines to promote program integrity.

1. Maintain a master copy and at least one backup or reserve copy of all approved material in the SDL. Retain the master copy under the best physical security warranted by the development circumstances.

The master-copy materials should not be used for testing or updating, except in cases of extreme emergencies, and then only while in the personal custody of personnel authorized by the project manager.

2. Make backup or reserve copies available for testing, update, and other development activities.

In the event backup copies are lost or destroyed, the master-copy custodian will create replacements from the master, exercising every precaution to protect the master.

3. In the case of master copies of magnetic tapes or removable disks, then in addition to the current master, retain at least the previous master

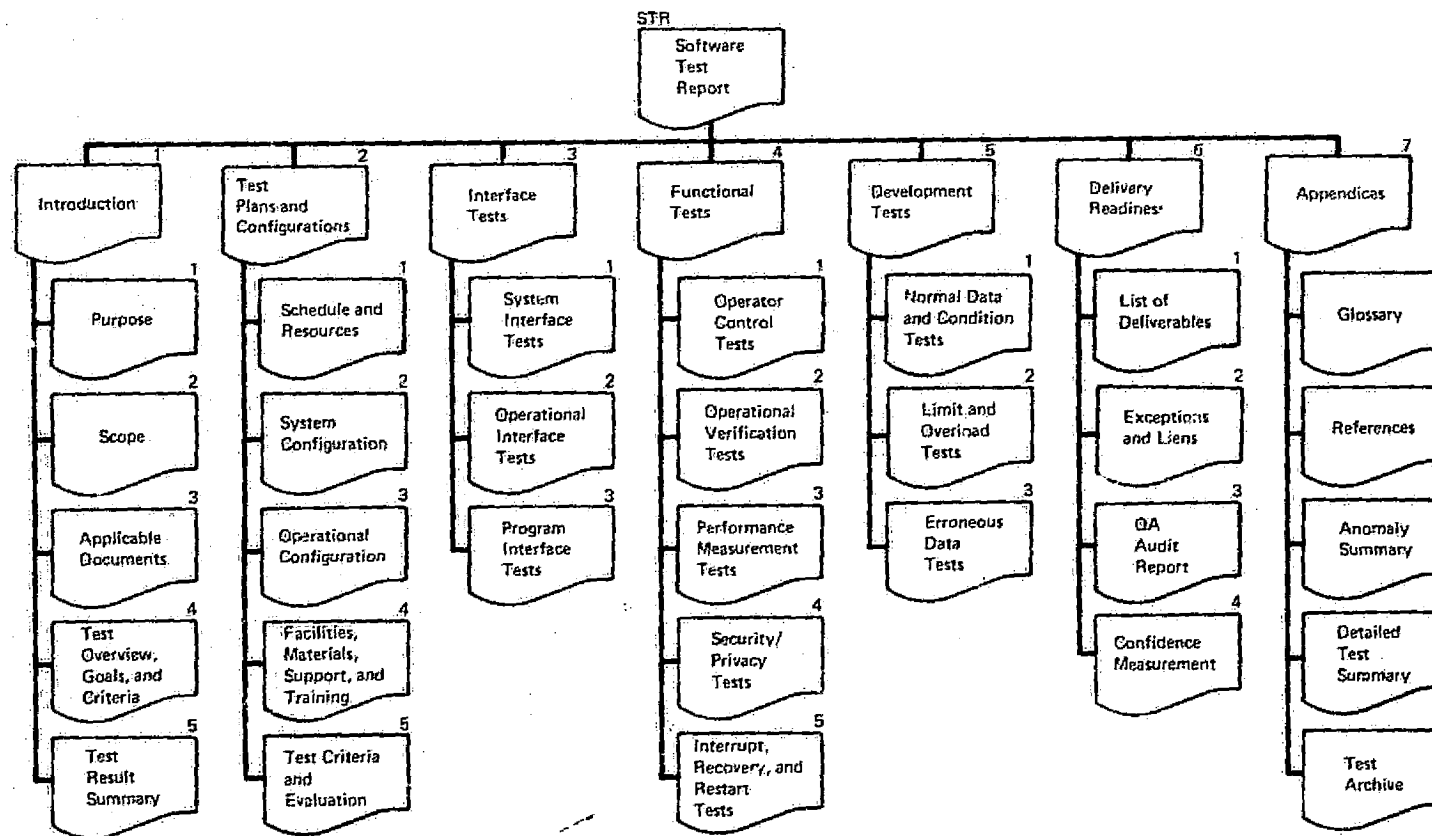


Figure 15-15. Graphical outline of the Software Test Report

and all the update transactions in a form to recover the current master, should it be lost.

4. Keep all approved copy material up to date on a regular, workable update schedule.

15.8 SUMMARY

Even if any one project does not specify or require every procedure described in this chapter, nevertheless, I have presented what I feel is a total and unified methodology for assuring the quality of delivered software. I have tried to do this by describing QA functions rather than QA personnel or organizational assignments.

I have tried to integrate QA into the day-to-day software development activities to promote consistency between the software requirements and the delivered product, and to detect problem areas early in the implementation effort, thereby lowering design risk.

REPLACING PAGE BLANK NOT FILLER.

XVI. LEVELS OF DOCUMENTATION

The word "documentation," as chiefly used in this chapter, refers specifically to information recorded during the development of a software system to explain the pertinent aspects of that system. The entire history, from requirements and program definition to design, to coding, to checkout and verification, and, finally, to certification, may need to be documented in one form or another. Included among the items to be so recorded are the purposes, methods, logic, rationale, relationships, capabilities, and limitations of the program components. Also, perhaps, to be included are reports of manpower, budgets, schedules, and implementation planning. User documents, planning documents, and the others in Figure 2-12 are further recordings of a program's pertinent aspects that must be written.

Obviously, some programs and projects will require more or less documentation than will others, and the documentation that is supplied may very well place a different emphasis in each of a various number of areas. The level and orientation of documentation depends on a number of factors, which includes program size, usage, intended life span, criticality of system interfaces, training of operators and maintenance personnel, and development team organization, to name but a few.

The objective of this chapter is to provide a uniform set of guidelines for specifying computer program levels of documentation to fulfill human communicational needs, subject to economic and schedule considerations. That is, the guidelines are aimed toward producing cost-effective, as well as useful, documentation. The types of documentation to be produced and the procedures for doing so have been adequately discussed elsewhere in this text and will not, therefore, be repeated, except, perhaps, by summary.

16.1 HUMAN FACTORS

Documentation is for *humans*, to communicate what humans need to know in order to interface in their intended way with the program. Documentation failing to meet this simple criterion results from inadequate human engineering. Problems involving documentation, thus, should be attacked as human engineering problems, not as programming problems.

Good documentation is characterized by order and form, which display a clear plan or design to whatever the writer wishes to communicate. Clear documentation does not fall into order by mere chance. Order results from careful arrangement of suitable materials to fit a definite purpose.

16.1.1 Problems Caused by Inadequate Documentation

A 1974 Report to the Congress [18] by the Comptroller General of the United States presented an analysis of over 710 questionnaires received from Federal ADP personnel and auditors from more than 70 computer installations throughout the United States, Europe, and Asia. This analysis showed that inadequate documentation had increased the cost of ADP operations, weakened management controls, contributed to the loss of funds and assets, and limited the potential for sharing computer programs, especially mathematical models.

The report further cited inadequate documentation as being a significant factor when programs had to be rewritten and systems redesigned, when excess time was required to make modifications, and when delays were encountered in completing assignments. Some of the deficiencies reported were: that operating instructions were not supplied, or were not clear; that the mathematical model was not explained clearly; that sample runs were absent; and that flowcharts were not supplied.

In addition, it was reported that the lack of documentation made it difficult for auditors and managers to review and identify internal controls, which then required considerable expenditure of time from the programming staff to explain to the auditors and managers how the computer

system functioned. Lack of adequate documentation in many instances prevented the application of programs to situations other than those for which they were originally designed, but to which they were suited, if properly modified.

The net effect of inadequate documentation, concluded the report, was a high aggregate cost that could have been avoided, had proper documentation been provided.

The causes of inadequate documentation seemed to boil down to two lacks: The lack of standards and the lack of review and enforcement. Consistent preparation of adequate documentation under tight performance schedules requires good standards and continual review to ensure compliance with them, stated the report.

There was literally no disagreement among the respondents in the study that good documentation practices should be maintained for all programming projects. The problem seemed to be in assuring that the necessary documentation actually got done. There always seemed to be many competing tasks for the software system implementors' time, and documentation often took low priority—a choice which all too often was sorely regretted later.

16.1.2 Fulfillment of Documentation Objectives

Deciding the type of documentation and the amount of detail needed in each circumstance can be made an easier task (than impossible) by setting objectives and criteria for documentation relative to *who uses the documentation*, *what type of application they have*, *whether there is sharing potential of the program*, *how complex the program is*, *how often the program is expected to operate*, *how long its expected lifetime is*, *how much the documentation will cost in relation to expected benefits or probable penalty costs*, and *schedule criticality*. Some of the objectives of good documentation [19] are to:

- a. Aid workers in producing the program.
- b. Aid managers in monitoring and managing production.
- c. Enable and assist the user to operate and understand what is being done.
- d. Permit quick and effective modifications when needed.
- e. Form a basis for new system planning.
- f. Assist QA functions.
- g. Increase sharing potential.

Of these objectives, only the first is of direct interest to the programmer and the second is of concern to his supervisor. The rest are, in large part, downstream benefits. Hence, the real monetary value of good documentation is also likely to be downstream. Much of the documentation to be supplied in fulfillment of these other objectives, however, is generally required of the developer. Without proper standards, the developer is forced to guess what level, form, and content the documentation for others must take; without proper incentives, he may not be at all conscientious about his guessing; without enforcement of standards, he may, perhaps (inadvertently), not be able to respond adequately to the needs of others.

All too often, we tend to assume that anyone who can write a program is also capable of communicating to others how that program works. To make this assumption a fact is a matter of professional training and discipline. Program documentation techniques need to be studied and learned just as diligently as programming techniques do. As I discussed in Chapter 10, each worker in the development needs to be capable of describing his work so that others can read it. It's just good engineering practice.

I don't mean to imply by this that there isn't a place for professional software documentors in a software development. There is. Perhaps the best example of this is the documentation for users (or operator) of a program. Here, the ability to communicate is paramount; unless instructions are complete and effective, a program can rarely be used to its fullest capability.

Moreover, users of a program have a right to expect that the documentation they see will measure up to the same professional quality standards as the program itself.

I spoke earlier of incentives. There is little need for standards enforcement when standards are viable and accepted by each developer as his own self-discipline. However, programmers will supply most where their incentives are greatest. If a low priority is placed on documentation, or if the documentation effort is given insufficient time or budget, or if the documentation is not regarded as highly as the program itself, then the motivations for quality documentation cannot be expected to even tie for first place.

16.1.3 What Constitutes Good Documentation?

Documentation quality is characterized by four attributes [20]: completeness, accuracy, clarity, and economy. In the case of program

documentation, these are promoted in the team model in Chapter 10 by making required documentation be the interfaces between development activities. The quality produced is precisely that dictated by the activity receiving the documentation, as it must be adequate in order for that activity to continue. Any documentation supplied, but unused, is superfluous, and could have been omitted; any not supplied, but needed, is quickly apparent, and must be supplied before the activity can continue.

Good documentation addresses a *need* on the part of its intended readers. There are several factors that help match this need to the *usability* of documentation. Big, overly detailed documents, for example, generally tend not to be used very much, even though their users may claim they need those reams of detail. Size must not be allowed to become synonymous with complexity.

The *method of presentation* very often contributes to clarity. Graphical representations or displays and narrative descriptions are particularly useful in promoting understanding, for example. The proper narrative content and the proper graphic to be used, however, depend on the idea to be communicated. For program logic, flowcharts or indented procedure descriptions (such as CRISP-PDL) are effective; for routing of information, one may use data-flow diagrams; for data structures, a picture of the data layout, format, etc., can be illustrative; for real-time program interactions, there are timing charts, Petri networks, and state diagrams. All graphics require accompanying explanatory narrative to be effective. Other candidate tools that foster communication are:

- a. Standard outlines for documents.
- b. Standard symbols for graphics.
- c. Standard format for narrative descriptions.
- d. Use of cross-references and indexes.
- e. Display of documentation hierarchy.
- f. Display of program hierarchy (tier charts).
- g. Use of decision tables.
- h. High-level programming languages.
- i. Use of mnemonic descriptors.
- j. Use of comments in code (for items uncovered in SSD).
- k. Sample runs and other examples, with explanations.
- l. Timing analyses, response plots.
- m. Schedules, PERT Charts, other management graphics.

Another factor that governs quality is *orientation*. Documentation for a program must always assume that the reader have some prerequisite level of skill. For example, if a reader does not understand English, then he is not going to understand a program document written in English, and nobody can blame the documentor for that. Similarly, no one can expect a person who does not understand FORTRAN to fully understand the rationale for certain operations in a FORTRAN program. Nor can one expect that a program to compute spacecraft trajectories can be made understandable to a person who has not passed high-school algebra. What one seeks in quality documentation, however, is an immediate recognition of the level of reader required and a match between the intended readers and those who need the documentation.

A third factor is documentation in the proper *level of detail*. If a system is of any size, different users will need different degrees of detail in the information they extract. The highest level should be readable by all and tell the reader whether or not the deeper levels are of interest (or are even readable by him). At each level, there should be a reference to the documentation above and below in depth—that is, there is a need for documentation to be organized hierarchically in detail and cross-referenced.

Documents also need to *display this structure* in a form that the reader can grasp and use as a roadmap toward finding the detail to be extracted. A graphical table of contents is often a very useful adjunct, except when structured flowcharts are being used; these display the program hierarchy very naturally, already.

Economic considerations and rapidity of accommodation must be folded into the choice of the *documentation media*. Documentation published as a bound book will not be flexible enough nor inherently fast enough to accommodate the SSD of a rapidly evolving system, but may be ideal in the case of a widely used, stable, user manual, for example. The need for quick answers to specific technical questions, when there is a maze of documentation that must necessarily exist for a dynamically evolving, complex system, also may tend to indicate that any conventional form of written documentation may be ineffective. Automated documentation, on the other hand, if produced and kept current by computer and viewed by way of a suitable set of terminals, may help cope with the situation, but the extent to which it can be made to contribute is limited by economic tradeoffs and user work habits. (A user who does work at home in off-hours will be discouraged from accessing documentation through a non-portable terminal, for example.)

In summary (Figure 16-1), for good documentation, the methods of presentation should match the concepts to be communicated; the orientation should be toward the intended reader; the documentation should be organized in hierarchic levels of detail, with an appropriate identification of the required aptitude of the reader for each; the documentation should be organized in a form easily displayed or grasped, to facilitate extraction of sought-for detail; and the media for documentation should be chosen so as to be readily available, updatable as appropriate, and conformable to the development environment. Such documentation must be as complete, accurate, and lucid as end-to-end economic developmental or life-cycle considerations will allow.

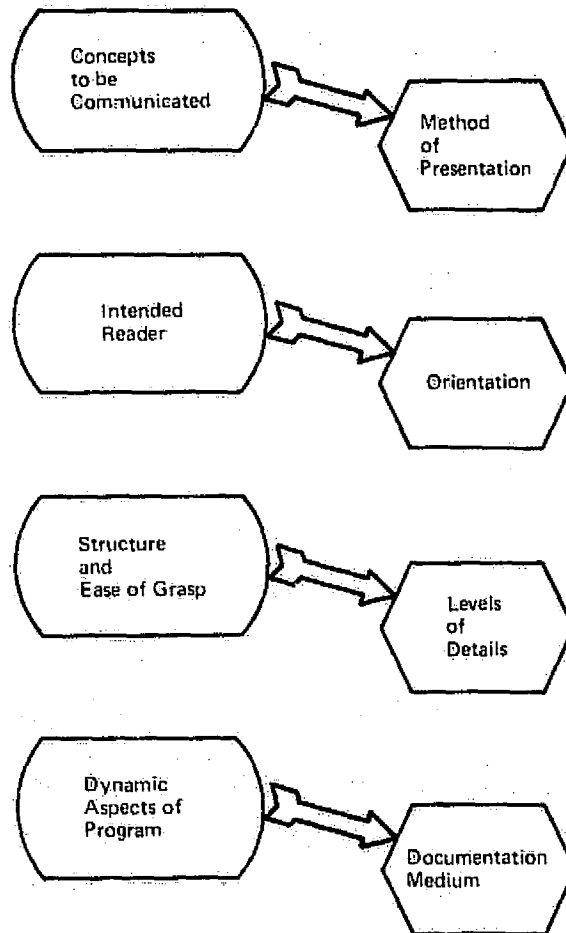


Figure 16-1. Some factors that influence the quality of documentation

16.1.4 Users of Documentation

Technical documentation falls roughly into five categories: (1) that which establishes the software requirement; (2) that which defines the program design guidelines, program architecture, and development resources; (3) that which specifies the detailed program implementation and testing; (4) that which instructs users and operators; and (5) that required internally during the development process. In previous discussions, I have focused on one such document in each category, viz., the SRD, SDD, SSD, user manuals, and the Project Notebook containing programming bulletins, decision summaries, etc.

Besides technical documentation, however, there are management and planning documents (the SRD and SDD are largely of this type, in addition to their technical content). The content of each document produced in an effective software development must be balanced between technical and non-technical information, based on the needs of the users of that document.

According to the congressional report mentioned earlier, 99% of the respondents indicated that they believed the needs of the intended users should be paramount in making documentation decisions. Because of the various types and levels of documentation needed across the user spectrum, such judgments often require subjective assessments of benefit.

Users of the program itself need clearly written, accurate, and meaningful descriptions of how to operate the program, create input, and interpret output. Programmers need computer system manuals and the technical documentation that forms the evolving software specification. Auditors need documentation that cross-references the entire development, so that they may verify that that which was required was specified, that which was specified was built, etc. Quality Assurance needs documentation in order to validate that deliverables are acceptable. The Software Development Library needs documentation of its configuration-control procedures, anomaly reporting measures, etc., as well as documentation of the software being produced.

Project management and programmers alike need documentation in the form of status reports, policy overviews, and team bulletins during the program development progress. Later readers of the program will need to know what the various parts of the program do, and why; they will need to see the relationships among data items, information flows, and storage configuration in graphic as well as narrative form.

Modifiers and sustaining personnel will need maintenance procedures, such as instructions for updating a data base, installing a new version of the

program into the system, handling anomaly reports, and making changes quickly and effectively.

Deciding on the type of documentation and the amount of detail needed in each circumstance to satisfy each different type of user need is not a clear-cut process. Nevertheless, standards can be prescribed that will aid development projects in making rational decisions relative to such questions.

Many may think that all the rules I have given in previous chapters for meticulously documenting each phase of development constitute "over-kill"—more than any project would ever want. Perhaps so. Some may even think there is a need for yet more. Perhaps so. In the remainder of this chapter, I shall address some of the factors that one should consider in determining what type and how much documentation is proper for a given development, and I shall discuss standard categories of documentation levels.

16.2 DOCUMENTATION STANDARDS

Documentation standards tend to coordinate system structures into a uniform mold. Standard outlines and formats, as well as standardized vocabularies, necessarily are based on assumptions made about the kinds of systems to be described. If these assumptions are improperly conceived, then the organization of the documentation may be poor for a particular application, and vocabularies may have to be stretched or misused.

Yet, standard formats and vocabularies allow the readers of documentation to find some piece of information without needing to learn the concepts and vocabulary peculiar only to one system or program. Because of this, such documents are easier to review, audit, and maintain. There is also less likelihood that designers and coders will overlook the full impact of statements or graphics contained within it. Thus, standards tend to improve the communication between all people involved.

To fulfill its goals, then, documentation standards must be well-conceived, flexible, and designed so as to make the fewest number of assumptions relative to system-dependent or program-dependent considerations. That is, documentation standards should be made to apply only to the extent that they are relevant to all (or a significant portion) of the programs being developed within an organization (if not within a discipline or an entire industry).

16.2.1 Standard Levels of Documentation

In large, long-life-cycle programming systems, documentation must be provided in considerable detail; however, for small, single-purpose, or "one-shot" jobs, hardly any detail may be needed at all. In important or widely used applications, documentation may be typeset with professionally drafted artwork; in small or exploratory programs, handwritten text and hand-drawn sketches may be sufficient.

In scanning the document outlines contained in the appendices, the reader may note that there is a lot of detail called for in each. Every one of the topics included in each of the outlines is of potential concern to every program being developed, albeit many topics can be dismissed immediately as invisible in the current application, or not applicable to it, or understood as a standard across many similar situations. There will often be additional topics needed in the outlines for some applications.

Requirements relative to classification of detail and characterization of the documentation medium should normally be settled prior to the initiation of work. Such requirements are easily stated only if there are well-defined, standard classifications for documentation from which to choose. The specified classification then becomes the basis for planning, directing, and controlling the documentation effort.

The remainder of this chapter describes general requirements for standard levels of documentation, and for application of these requirements to intended usages. These standards encourage the production of only those forms for documentation that are needed and adequate for the purpose.

The *level of documentation* of a particular type needed by a given project is defined as the characterization of that documentation by a classification of the detail required and by a categorization of the documentation medium and format quality. This grading can be fitted to the needs of readers and to the available development resources.

I shall first define four classes of detail, ranging from "Class A" (most definitive) down to "Class D" (least definitive). The requirements for each such class are relaxed successively for each lower class from A to D. I shall also, then, a little later, define format-quality categories ranging from "Format 1" (highest) to "Format 4" (lowest). Guidelines for selecting among each of the classes and categories also appear with each definition.

The spectrum of documentation levels defines a lattice structure (Figure 16-2), or a set of partial-ordering relationships according to detail class and format category. For example, a document graded A1 is both very detailed

and beautifully published; a document graded D4 contains little detail and may be handwritten.

Costs to produce documentation rise as the level of detail increases and also as the format quality increases, and such costs can be estimated or measured with fair accuracy. The costs of not having documentation of a given type are not quite as easy to pin down, as these costs depend on downstream life-cycle utility factors that may or may not happen to occur. Nevertheless, the agency that sets documentation requirements must carefully weigh life-cycle factors if there is to be a cost-effective documentation plan.

I must, therefore, assume, in the few guidelines I give here, that the schedule is realistic in that it plans for documenting to the selected level, and that sufficient funding has been allocated for document generation and distribution. The guidelines that follow are meant to help establish and select an appropriate level for documentation relative to other issues. The principal factors other than cost and schedule in selecting a level of needed detail are the (1) assumed levels of skill of the intended readers, (2) the sharing potential of the program, (3) its expected lifetime, (4) its complexity, (5) its use frequency, and (6) its generality of application.

I shall illustrate the concept of standard documentation levels more definitively using the Software Specification Document as a case in point.

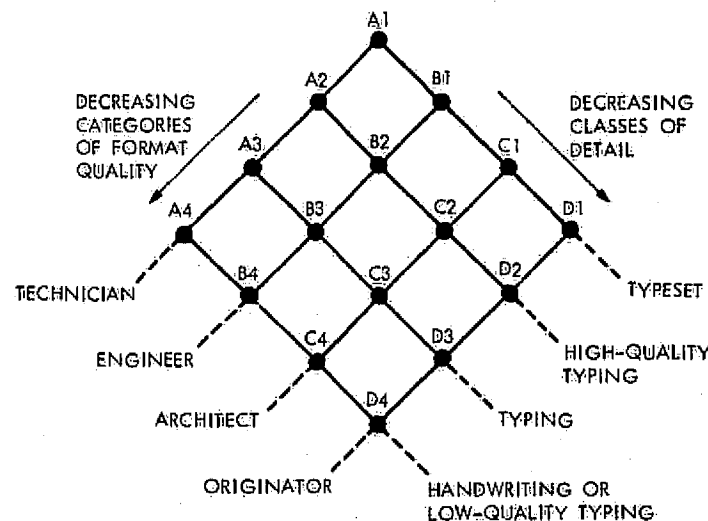


Figure 16-2. Documentation-level lattice (class of detail is graded A, B, C, and D; format quality as 1, 2, 3, and 4)

Suitable interpretations for categorizing other types of documentation (requirements, operations, etc.) are then not difficult to imagine.

16.2.2 Classes and Criteria for Detail in Software Specifications

The classes defined in this section signify criteria for the amount of detail to be provided when writing an SSD. These classes are consistent with NASA computer documentation guidelines [21], but are styled in the manner of JPL standard practices for engineering drawings [22].

16.2.2.1 Class A Specifications

Class A documentation is the most detailed; it contains specific definitions and detailed descriptions of every significant factor or item within the software specification, so that the program specifications can be understood, implemented, and maintained by any technically qualified personnel (technician or equivalent) without consultation. This class of documentation results when the project adheres rigorously to the SSD documentation rules of Chapters 11 and 12. All descriptions, functions, and operations are specified in that level of detail which permits a correctness assessment by the designers on an individual module basis, and then coding without functional or algorithmic ambiguity; no factor of an item contained in Class A documentation is left to the discretion of the implementor. All operations within unstriped flowchart symbols (or other suitable specifications) are covered by appropriate references to published works, external standards or internal conventions, or else are at the programming language level (see Figures 12-15, 12-24, and 12-25 for examples).

Since the SSD is also used as a maintenance document, a Class A specification could also be oriented merely to describe all the pertinent aspects of the system, its operational environment, its interfaces, testing, and external data bases, all to the level of detail that permits an experienced maintenance programmer to correct errors or implement authorized changes without excessive expenditure of time to understand the program, or without the need for consulting the program developers.

This class of highest detail is appropriate for SSDs whenever the intended readers/users are to be relatively unskilled (technician level) personnel, and need such detail to perform effectively, or where there is a need to ensure that certain particulars be interpreted in a specific way. In some cases, such as maintenance documentation, there may be a cost advantage in lessening requirements for Class A detail by employing personnel with higher levels of skill. However, since it is not always possible to select or predict the community of readers or users, then in such cases, Class A detail may still be appropriate.

This class is also appropriate for SSDs whenever a detailed audit of the program is required, or when the SSD is to be used as a contractual instrument with minimal contract coordination and review. Other indications for selecting this class are (1) a critical application, perhaps involving personal physical risk; (2) good sharing potential; (3) high frequency of use; (4) highly intricate concepts to be communicated; and (5) long or continuing program life cycle.

This level of detail probably finds its most applicability in user manuals, and rightly so: The writer of a user manual is generally unavailable for consultation, so the user needs the extra detail.

16.2.2.2 Class B Software Specifications

Class B documentation requirements are essentially the same as for Class A, except that the requirements for item definition detail are somewhat relaxed. Class B documentation, however, is to be suitable for conversion to Class A quality by the addition of further detail, without extensive effort on the part of the supplier of that detail.

Class B specifications define every factor of the software item being described to the extent that qualified personnel (engineer or equivalent) using documented techniques and approved programming practices can satisfactorily produce that item entirely from information supplied. Some specifications for coding of functions, operations, data structures, etc., may be left to the discretion of the implementors if these will satisfy program requirements with respect to performance and quality without unreasonable risk (see Figure 16-3 for an example). Assessment of control-logic correctness must be possible on an individual module basis.

The level of detail required for SSDs oriented primarily toward program maintenance also drops, to that amount needed by qualified maintenance personnel to correct errors or to implement changes, either after a minimal consultation with the program developers or after some reasonable review and head-scratching, to discover how a specific part of the program works.

Class B detail applies whenever readers are assumed to have more skill (engineer level) than that deemed appropriate for Class A, or are otherwise capable of responding properly to less detail. The class is also appropriate for specifications whenever the audit requirements are for consistency only, or when the SSD is to be used as a contractual instrument with moderate vendor coordination and review, or with limited contractual risk.

This class applies to specifications for normal applications programs with normal usage, but with limited sharing potential. There may be few intricate concepts to be communicated, but there is expected to be a long or continuing need for the program and its documentation.

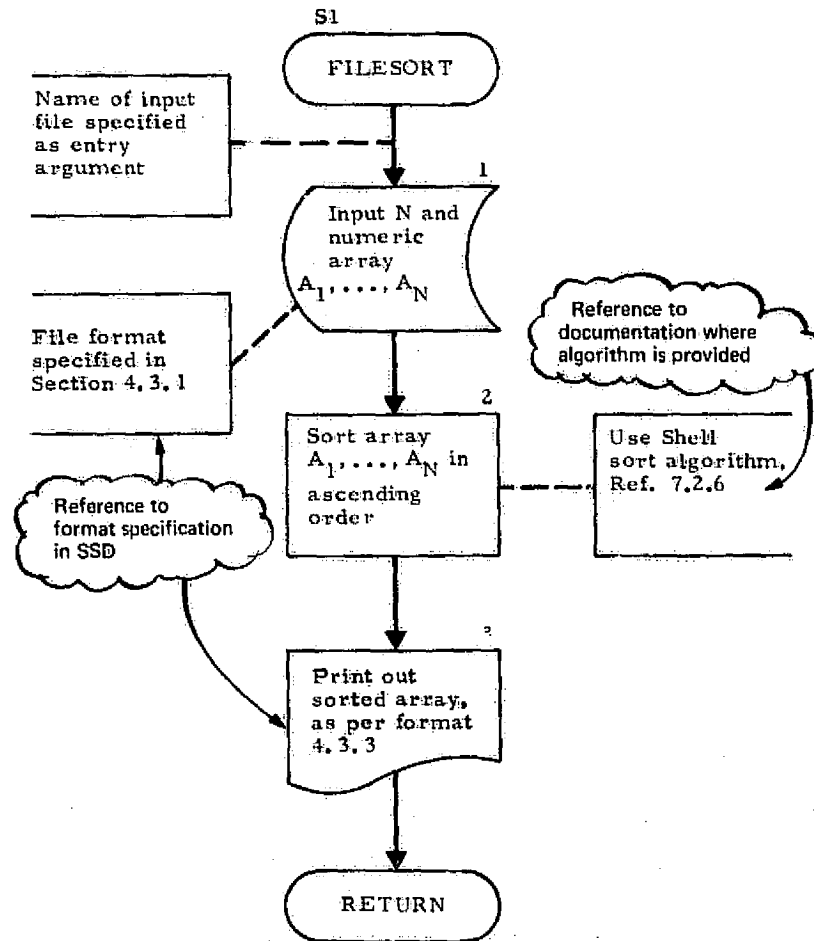


Figure 16-3. An example of a Class B detail flowchart (text within clouds is not part of the flowchart, but are explanations of flowchart conventions; figure also demonstrates Format 2 quality)

16.2.2.3 Class C Specifications

Class C documentation represents a yet further relaxation of requirements for detail than does Class B. Class C documentation may require a considerable effort in rewriting to supply detail to meet Class A or B standards. Class C documentation of design detail need only extend down to that architectural level sufficient for skilled programmers using hierarchic, modular, structured programming practices to produce an acceptable program. Callouts for standard algorithms, operations, etc., may be used in Class C specifications. The specific methods are left to the discretion of the programmer, subject to approval by the designers, and provided there is

minimum increase in risk in not satisfying program requirements with respect to performance and quality and, perhaps, at a moderate increase in debugging time or exploratory coding (Figure 16-4). However, control-logic correctness must still be determinable on an individual module basis.

Class C documentation also further reduces the requirements for detail supplied to maintenance programming. The use of Class C documentation may require more extensive consultation with program developers, or a more extensive analysis or reworking of certain parts of the program by the maintenance personnel to correct errors or to make modifications. The minimum documentation required is that necessary to set up the program source medium for operation and modification: I/O formats, setup

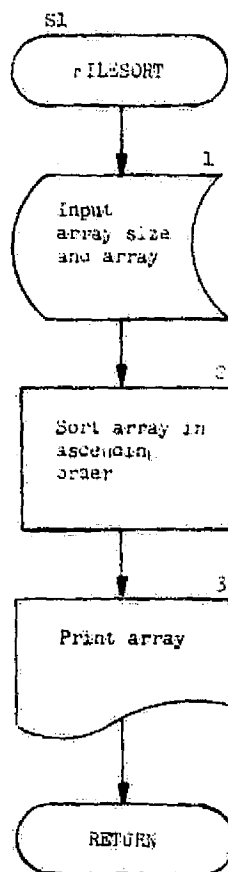


Figure 16-4. An example of a Class C detail flowchart (corresponds to the subprogram in Figure 16-3; Format 3 drawing quality used for illustration)

instructions, and the liberal use of comments in the source listing (for items not explained in the SSD; however, redundancy is allowed if ease of use is enhanced).

Class C should be chosen to document programs at the architectural or feasibility level, when there are no formal requirements for QA or audit. Its use as an SSD should be limited to cases involving implementation by highly skilled personnel within the cognizance of the project manager; it generally may not be acceptable or satisfactory as a contractual instrument without close coordination and review of contractor performance.

Class C documentation is probably also advisable for uncomplicated programs with anticipated low usage, no sharing potential, and short life expectancy. It may also be considered appropriate for documents within stable organizations having low attrition rates, or having high levels of expertise and experience.

16.2.2.4 Class D Specifications

Class D documentation is the minimal acceptable level of detail advisable for any program whose documentation is meant to be retained and, perhaps, read by others (or by the implementors, at a later time). Such documentation should only be deemed acceptable in cases where no upgrading of the documentation class is anticipated, as it may be generally unfeasible to upgrade the classification to Class A, B, or C without a concerted redocumentation effort. Standards for minimum detail are at the discretion of the preparer's supervisor.

Programs documented as Class D are suitable for maintenance by the original implementor only. A conceptual example of a Class D flowchart box appears in Figure 16-5.

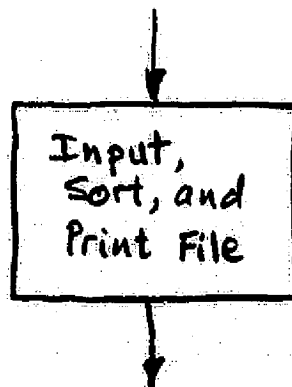


Figure 16-5. The FILESORT subprogram represented in Class D detail (also Format 4)

This class of least detail should be limited to summary material, overviews, and other reports of minimal complexity where there is a need to record capability, work done, results, or what a program does for historical purposes. Class D documentation probably applies to "one-shot" or single-use program SSDs, or to SSDs for programs requiring under one man-month or costing under \$2000 (these are not meant to be equivalent, see [21]).

Class D documentation should only be considered for specifications in those cases where the implementor is well-informed of the function and use of the program, as well as system, environment, and other implications. The use of such Class D documentation should usually be restricted to use by the developer only, and may necessitate extensive verbal contact with other readers, or extensive time, revision, or rework (due to lack of understanding) by those, other than the developer, who wish to correct errors or to make modifications.

16.2.3 Categories and Criteria for Format Quality in Software Specifications

The categories defined by this section delineate format and publication quality standards for documentation of any detail classification, as covered by Classes A, B, C, and D. Combination of class and format specifications are commensurate with NASA documentation guidelines [21], but are more flexible, as the criteria for selection are herein made as separate, independent issues.

16.2.3.1 Format Category 1: Formal Publication Quality

Format 1 generally applies to the documentation of programs that are of sufficient general interest, wide usage, or organizational-image value so as to be announced and distributed in the highest publication quality available. Such documentation should be prepared in a formal, rigorous manner, with in-depth technical review, meticulous proofreading, full editing by professional documentation personnel, and organizational approval for release and distribution.

In many cases, the text of such documents will be *typeset* and permanently bound, or of comparable quality; all artwork will normally be of professional drafting quality [23] equivalent to inked drawings with high quality lettering, suitable for "textbook" illustrations. The style is exactly that specified in Chapters 14 and 12 for narratives, flowcharts, and other descriptions. Flowcharts are further covered by Appendix B.

Alterations are distributed as errata if minor, and as revisions by reprinting and reissue if major.

Format 1, or Formal Publication quality, generally applies to cases where the document will find high usage, perhaps external to the organization, where the organizational image plays a role, where professional documentation personnel services are available for formatting, editing, composing, etc., where a general and wide-spread interest exists in the program, and where the program is stable.

Format 1 documentation probably applies most often to user manuals, announcements and summaries, and, perhaps, user-group bulletins. It would rarely, I think, be used for publication of an SSD, and probably should never be used for an SRD or SDD.

16.2.3.2 Format Category 2: External Report Quality

Format 2 requirements are the same as Format 1, except that the requirements for editing and composition are somewhat relaxed. Format 2 generally applies to the documentation of programs that are expected to be widely used within an organization but may also have some outside readership as well. Such documentation should be prepared with adequate technical review, good proofreading, editing sufficient to assure format consistency and clarity of expression, and organization approval for external release and distribution.

The text of Format 2 documentation should be of *high typewritten quality* and suitable for offset printing, such as that obtained using a 10-point IBM Executive Modern typewriter font. Illustrations and artwork should be drawn to professional ink-line drafting standards [23], perhaps with typed-in lettering (see Figure 16-3). The style is the same as that specified for Format 1. Alterations to the documentation are distributed as change pages, unless the document is permanently bound, in which case, changes are handled as in Format 1. In any case, the document is normally enclosed by protective covers.

External Report (Format 2) quality is the normal level of publication quality for documentation that is expected to find high usage, but where the aid of professional documentation personnel is limited, where the organizational image is less sensitive, where the general interest in the program is less, and where the program documentation is perhaps slightly less stable than one documented as Format 1.

Format 2 documentation probably applies to most of the user manuals, announcements and summaries, and user-group bulletins printed for use within an organization, or externally on an interim basis. Use of Format 2 for an SSD will probably only be feasible for highly stable programs, or those whose artwork can be computer drawn and whose text is contained

in, and reproduced by, a computer. Format 2 may also be suitable for some SRDs and SDDs.

16.2.3.3 Format Category 3: Internal Report Quality

Format 3 is a further reduction in report quality from that required for either of Format 1 or 2. Format 3 documentation generally applies to special-purpose or in-house programs that, after careful consideration of the possible interest of others, appear to have insufficient usage, sharing potential, or life expectancy to warrant a higher quality format. Such documentation should be prepared with project and QA review. Internal release and distribution are at the discretion of the project manager.

The text of Format 3 documentation should be *typewritten*, although there need not be any requirement placed on the typewriter font. Artwork may be hand drawn (in pencil, if reproducible) using standard templates and typed-in lettering. (See Figure 16-4 for an example.) This format should at least satisfy microfilming standards. There is no relaxation on the style of narratives or illustrations, however, from Formats 1 and 2. Any reproduction medium and binding suitable to the limited distribution is permissible. Covers are optional.

Alterations to Format 3 documentation are handled by distributing change pages or errata.

The normal working level of documentation within an organization is Format 3. It is generally used when the numbers of users of the documentation are limited, but where there is a need for continued use or a permanent record of the recorded items. The documentation is usually prepared within the implementing organization, without the aid of professional documentation personnel, and where there are limited funds or facilities for drafting and other artwork. The format, being less restricted, can also accommodate a somewhat less stable programming environment.

Format 3 documentation is applicable to the working-level SSD, SDD, and SRD, as well as low-use documents used for program maintenance and operations.

16.2.3.4 Format Category 4: Minimal Report Quality

Format 4 is the lowest quality of documentation and the least restricted. Review, QA, and distribution are at the discretion of the developer(s).

Format 4 documentation may be *frechand* (Figure 16-5), or may deviate from style requirements and standard practices discussed elsewhere in this text to whatever extent practical, at the discretion of the preparer's supervisor. Format 4 documentation need only meet the minimum requirements necessary for storage and retrieval in the SDL, if such facility

is even used. (It is usually desirable to keep on file for some period of time the documentation that results from program development, such as a program abstract, the Project Notebook, a compiled source listing, test cases, run examples, etc.)

Alteration methods are at the discretion of the preparer or his supervisor.

This lowest quality of reporting format, like Class D detail, most often applies to single use, "one-shot" jobs of minimal complexity, but for which there is a requirement to report or record what type of work was being produced, or what the results of a given effort were, and, thus, to retain certain information about a program for historical purposes. Format 4, thus, is suitable for exploratory, or look-ahead efforts, where there is little distribution potential, or for programs requiring under one man-month of effort, or costing under \$2000 (these are not meant to be equivalent, see [21]). Because of the informality of the quality restrictions, Format 4 documentation can probably be used better than other formats when there are unstable elements in the system.

Format 4 documentation is generally not suitable for most projects leading to operational programs, or for any forms of program documentation, except, perhaps, project bulletins, status reports, memos, etc.

16.3 PREPARATION OF DOCUMENTATION

The procedure for documenting a standardized development consists of supplying the required information in various phases, according to rules contained in the previous chapters of this text. Consistent preparation of adequate documentation under tight performance schedules requires standards or guidelines and continual review to ensure compliance with those guidelines. That which is going to be produced should be kept to the minimum necessary to meet the needs of those whom the program will impinge, because of the high costs of documenting software developments.

16.3.1 Rules for Project Management

Besides creating program documentation, gathering and recording status information during development is generally considered [10] to be one of the most important project management activities. However, there are no hard-and-fast criteria universally considered to be most important in evaluating the worth of a particular report or information gathering technique. Nevertheless, timeliness, accuracy, and definiteness (providing enough information to identify the accomplished work unambiguously) have always ranked high with many managers, and certainly do so with me.

Informal approaches to reporting status seem to work best; written reports should always be kept to a minimum. When used, they should be strictly constrained by length, time deadline, and very hard-headed analysis of their purpose. Defining and tracking concrete events based on functional capabilities or milestones, as exemplified by WBS reports mentioned in the previous chapters, are particularly useful to higher-level managers.

The following rules are guidelines to aid project management in setting standards for documentation and reporting [8].

1. Prepare a list of documentation responsibilities, specifying the individual responsible for preparing, coordinating, reviewing, and editing each document from its origination through its formal release and updating. Keep this list current, with updates throughout the life of the project.
2. Establish a documentation plan as early as possible in the program life cycle. Identify the documents by title, purpose, and criteria for content (see Section 16.2). Give required or estimated initiation and completion dates for each, and assign personnel through the list in Rule 1, above.
3. Establish the scope and general outline of the documentation plan. Address the relative weights or priorities to be given to technical documentation and to management documentation.
4. Set standards for the formats to be used, the time phasing to be followed, and the disciplines to be imposed in producing the documents. Then see that these standards are adhered to by adequate review.
5. Account for software documentation costs and effort adequately in manpower profiles and scheduling from the outset, and verify that orderly documentation is being carried on concurrently with design, coding, and testing of the program.

Quantify the value or benefit of documentation through the probable costs that may be required if adequate documentation is not prepared. Strike a balance between documentation costs and expected usefulness.

- 6 Estimate the probable sharing potential and needed availability at inception, and base documentation requirements upon this potential. If the prospects for sharing change, then alter the documentation and availability requirements appropriately after careful consideration.

A word of caution: seemingly one-shot or temporary programs have a way of evolving into frequently used programs. Increased sharing potential can result in lower-

cost future developments, provided that recognition of the potential is made early enough for proper documentation and announcement provisions to be made. Even subprograms within a temporary or one-shot program can often have sharing potential if properly identified. However, it is probably not cost-effective to consider sharing low-cost programs or subprograms, except in special cases.

7. See to it that the project documentation requirements and documentation plan are understood by personnel responsible for document preparation, distribution, review, approval, and updating.

8. Plan for the announcement, publication, and distribution of documentation to intended or potential users.

9. Institute incentives for documentation commensurate with the priority given to documentation in the project.

10. See to it that documents and status reports are kept current and visible to all pertinent team members through the SDL.

11. Use personal contact and meetings to coordinate policies, decisions, and interfaces between adjacent parts of a system. Confirm and record agreements after-the-fact in written form. Ensure that work based on such agreements proceeds from the written documentation, not on recollections of participants.

12. Insist that documentation be contributed at regular intervals into a project master file in the SDL, in addition to those files retained at the working level. Verify that there are adequate provisions for updating both types of files and for querying these files for status report generation. (If such files can be computer based, so much the better.)

13. If documentation is not up to standards and continually seems to be troublesome, *stop all activities not related to documentation and bring it up to standard.*

14. Insist on simplicity, directness, clarity, and quality of expression in documentation in as brief a form as the assigned level of detail permits. Only when this can be achieved can "more documentation" be made to mean "better documentation."

16.3.2 Rules for Documentors

The following simple rules define the responsibilities of documentors.

1. Understand thoroughly the project documentation requirements and the documentation plan.
2. Understand the intended use of the document and the needs of the intended users.

It may help to make a list of all personnel groups who form the audience for this document, and to imagine yourself in the role of each of the readers. Clarify what each such reader would want in the sort of documents being prepared before writing the document. If necessary, interview representatives of the reader groups.

3. Understand the overall content and appropriate level of detail required for the document.
4. Verify that applicable standards have been selected for document format and content.
5. Supply documentation that satisfies project requirements, fits the documentation plan, communicates with the intended users to the extent prescribed by the level of detail, and adheres to specified standards.
6. Submit documentation for review, correction, and amendment in phases concurrent with other software development activities.

16.4 SUMMARY

Adequate documentation of computer programs is clearly an essential element of efficient and economical use of computer systems. Good documentation prevents waste and unnecessary costs in many ways—by making program modifications feasible, by making redesigns easier, by making internal controls work better, by facilitating the work of auditors, and by a host of other ways, all equivalent to making programs usable by others.

Lack of needed documentation and low-quality documentation are problems in human engineering, which in large part can be remedied by setting good standards, and then seeing to it that work necessary for documentation is performed according to these standards.

I have outlined standard levels of documentation by content and format, and I have given typical criteria for choosing among these documentation levels. Detailed rules for flowcharts, narratives, information flow, decision

tables, etc., are contained in each of the preceding chapters pertinent to each development activity.

XVII. A STANDARD SOFTWARE PRODUCTION SYSTEM

As computer applications continue to advance, there is an increasingly more intense need for better and better tools to aid in the construction of programs. The more difficult a task is, the more powerful the tools must be in order to combat the burden of complexity that bears down on implementors.

There are certain jobs associated with developing software that currently, or, perhaps, inherently, humans do better than computers, such as creative thinking, problem solution, and subjective judgment. Previous chapters in this work have set forth standard, structured disciplines that attempt to organize such processes, and thereby enhance individual and team productivity.

There are, on the other hand, many software development tasks of a more predefined computational, routine, or clerical nature that computers can do quickly, easily, and perfectly, but which humans tend to do in a much slower, more awkward, more difficult, and error-prone manner. Suitable automated tools to reduce this routine or clerical burden during

software generation can significantly reduce programmer time and effort, can yield more reliable programs, and can produce enhanced documentation. In addition, the wide use of automated development support tends to standardize programming methods, programming structures, program documentation, and all other areas for which support is provided.

The purpose of providing automated support is to make it as easy as possible for the humans engaged in a software development to perform those functions which only they can provide, and yet deliver all that is expected in the way of quality programming, proper documentation, adequate management visibility, and good configuration control.

This chapter discusses, therefore, the standards for a well-formed, efficient, and effective program production support system, in which both human and automated tools play a well-balanced, integrated part.

17.1 AN INTEGRATED SOFTWARE PRODUCTION SYSTEM

Science, technology, industry, and commerce are based on the ability to measure things or phenomena, to describe the results in numerical terms, to make comparisons, to make decisions based on these comparisons, and to implement these decisions using standard means. In order that the software industry be a stable member of the industrial family, its products, too, must be describable, measurable, and implementable in standardized ways. Predictability of function, performance, cost, size, and other qualities, then, becomes a characteristic of the industry. This is what technology is all about, and the establishment of sound production tools into a standard system does much to bring the development of software products into such a technological arena.

The bounds of a programming system are not limited merely to the set of automated capabilities one may find in a computer library. Rather, a well-engineered production support facility must consist of a number of other elements taking various forms.

17.1.1 The Elements of a Standard Production System

The first element required in a good production system (Figure 17-1) is a body of sound, effective methodology for systematic development of the software items. Such methodology is produced as a result of dedicated study into the problems of software over an extended period of time. This methodology is continually evolving, in cycles, as new discoveries are made and as amendments and modifications to existing technology are

introduced. Directions for change come about through observing the effects of changes in previous cycles, through evaluation in pathfinder projects, and through continuing applied research.

The second component of an effective programming system is a set of standard practices, human production disciplines, and software user manuals, together with a user training program, which identify and formally establish the adopted technology and transfer it into a regular implementational and operational environment. This set of elements also

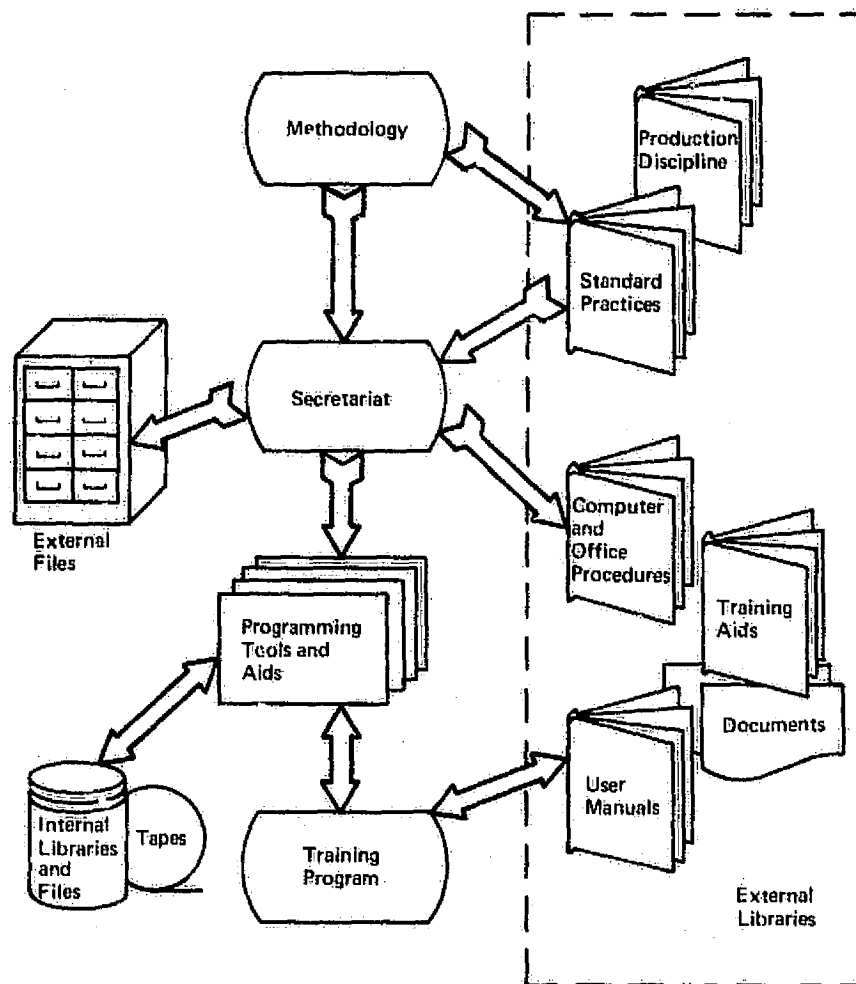


Figure 17-1. The elements of a Standard Production System

serves as a concrete point of reference against which firm evaluations of the established technology by the user community can be made toward possible future modifications.

The third component of an effective software production system is a central service facility, or programming secretariat. This facility supports the production of software in a number of various ways, such, as by serving as a distributive agency for standard practice instructions, manuals, and other production materials; by providing data-preparation or data-entry services; by updating and editing textual material, e.g., program code, documentation, and other material as contained in, and controlled by, the central facility; by generating or updating standard graphic material held in project files using standard automated graphics capabilities; by perhaps administering the software change-control process; etc.

The first three parts of the programming system, then, are oriented toward identifying and solving the human factors that surround software developments. These have largely been the subject of this monograph up to this point, and do not need to be further expanded upon here in any great detail. The fourth and remaining part of the system consists of a set of automated aids oriented toward reducing the time humans spend (or should be spending) producing adequate documentation, status reports, and the program itself. These aids are, in fact, the main subject of the remainder of this chapter.

This fourth and final part of the programming system, thus, consists of a library of standard languages and processors, automatic production aids, and management-support software, plus the necessary user documentation and training in the application of such tools to the task at hand. A full complement of software items in this category is difficult to imagine; however, as a starting point, the items in addition to a normal complement of coding languages should include facilities to aid in the design, testing, validation, and documentation (both narrative and graphic) of programs and data, and for gathering and reporting of management information, such as costs, schedules, current status, productivity, etc. Further, provisions for program and data base configuration management and change control, as well as features that permit quality assurance metering of the project deliverables, should be included.

The support facility includes a reference document library in addition to the computer program library above. The computer library contains a set of utility programs and subroutines that are available for use by projects developing software. The reference document library, then, contains the needed documentation for users to be able to apply these utilities and subroutines to their projects. In addition, the reference document library

contains standard practice instructions, development disciplines, programming textbooks, and other source materials needed by the community of program development and implementation personnel. Since creation and maintenance of such a library is a support function, its administration falls within the purview of the programming secretariat.

17.1.2 Classification of Implementation Aids

Automatic implementation tools fall into three categories, which I shall refer to as primary, secondary, and tertiary (Figure 17-2). Primary tools are those facilities absolutely needed to build a program in a practical sense. These facilities include the compilers (or assemblers) for the languages being used, program (linkage edit) loaders, data and text editors, and, usually, an operating system. Without these elements, the programming task is untenable.

Secondary development support aids are those tools which can be used to increase software productivity and/or program reliability significantly, but which are not needed in the absolute pragmatic sense, as are primary tools. These are tools that are justifiable and feasible economically whenever their implementational, operational, and sustaining costs are more than counterbalanced by the savings they produce. Since the allocation of

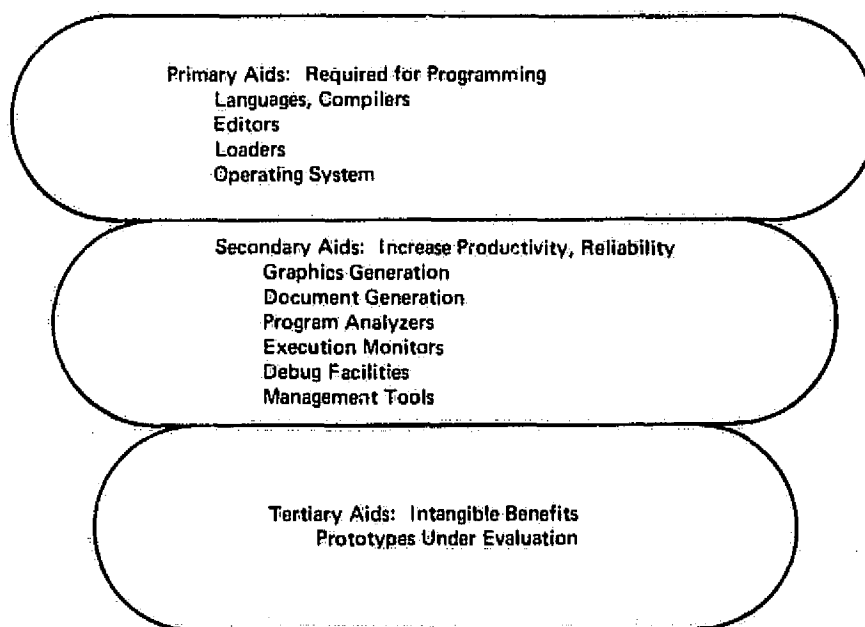


Figure 17-2. Categories of Implementation aids

development resources to the generation of new support tools can impact ongoing project delivery schedules as well as their dollar costs, the production of new tools may not always be advisable when these are in series with critical-path tasks, even though the eventual availability of such tools could show a later positive economic benefit. Such tools should always be carefully justified in terms of projected life-cycle cost savings through increased productivity, in terms of schedule impacts with ongoing projects, and in terms of software quality requirements relative to reliability, manageability, level of documentation, etc.

Tertiary aids are automated tools of less significant supporting value than secondary-class tools. Tertiary aids are those with intangible economic or program reliability benefit within a given production system. These tools could conceivably be justified on some basis other than economics, schedule, or program quality; however, that basis should be closely scrutinized before the tool is allowed to become a formal part of a "standard production system." Tertiary tools may, however, appear from time to time in a standard production system on an evaluational or trial basis, either to be upgraded to secondary (or even primary) status if the evaluation is favorable, or to be dropped if the evaluation is unfavorable.

In specifying a set of support tools for software development, the needs, desires, and frustrations of personnel must be considered in addition to economic and schedule factors. Moreover, the tools should exhibit human engineering qualities so as to be effective, generally applicable, and easy to use over a wide spectrum of applications within the standard system.

When a set of tools is implemented, each must be absolutely correct, validated to the point that it can be used with 100% confidence. How else can one use such a tool to validate other software products or to enforce standards, unless the tools themselves are faultless?

A final criterion I would impose is the near-absolute machine independence of the programming system application-user interface across the entire spectrum of host systems being used, within a minimal set of conventions imposed by the host hardware and software environment. Such conventions as log-on procedures, interactive vs. batch operation, file naming conventions, tape-mounting protocols, complement of peripherals, and word-length restrictions are very likely to vary from host to host; nevertheless, automated tools should be able to respond to these different environmental constraints and yet maintain effectiveness. Thus, while materials produced in one standard production system may not be absolutely portable in source form to another such production system, there should, nevertheless, be a large base of commonality and uniformity in the

products of each, with absolute portability of applications programs being the goal.

When user interfaces cannot be made to support absolute portability, they must, as a minimum requirement, support program mobility. That degree of portability vs. mobility sought is one that should balance any increased running costs of user programs against the decreased production, conversion, documentation, and other constituent costs during the total software life cycle.

17.1.3 The Programming Data Base

Automated or not, the production of a piece of software utilizes a data base, consisting of design information, coding information, testing information, management information, and documentation. Without automation, a great deal of time is spent locating, transforming, and extracting information from this data base to create meaningful views of the items extracted.

If automated, the complete data base could well reside in an integrated set of files, accessed and manipulated by standard processors. The computer system then becomes the design medium, the documentation-composition medium, and the software management medium, in addition to being the more usual coding, debugging, testing, and operational media.

Each element of a programming data base is generally interdependent on the others, whether automated or not. But if automated (Figure 17-3), for example, procedural design information can conceptually generate flowcharts or data-flow diagrams, if needed, and can be automatically output with the corresponding narrative; code can be audited automatically for conformance with standards and with the procedural design; project status information can be collected, such as numbers of modules completed or stubbed in design, code, or test activities, frequency and type of errors committed, etc.; test design criteria for traversing each program flowline can be identified; editing and updating of design, code, testing, and documentation can be controlled; all accesses to the programming data base can deposit activity information back into the management data base for project status reports generation; and configuration management standards can be enforced.

The programming data base is, therefore, one of the key elements in the overall concept of the standard software production system. Combined with its level of access, it forms the basis for recording and storing of programming data, provides the vehicle for the organization and control of a programming project, serves as the means of communication between developers, and interfaces program development personnel to each other

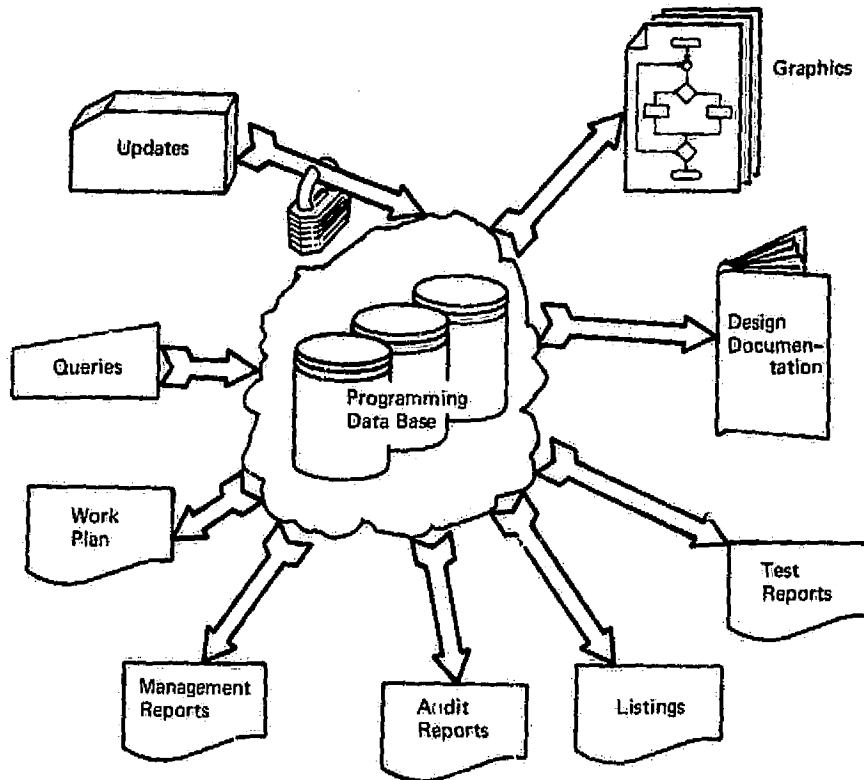


Figure 17-3. The programming data base in a production system contains all information from which documentation, reporting, and execution emanate

and to the computer. The principal management function of the data base is, then, to maintain a current and exact configuration of a program under development for all viewers. Further discussion of management data processing appears in Section 17.7.

17.1.4 The Programming Secretariat

The programming secretariat is an individual or group of personnel who support all development projects within a certain purview, much like a secretarial pool aids office management by performing a body of common, centralized clerical functions for a number of separate, but similar, users (Figure 17-4). Some of the functions typically performed by a secretariat were mentioned earlier: data preparation and data entry services, distribution of manuals and production materials, document preparation and revision, management reporting, archiving, etc. In addition to these

functions, the secretariat probably must also have the means to generate new elements and maintain existing capabilities of the software production system, to control the configuration and integrity of that system, and to collect user feedback relative to the functioning and use of that system.

Each software development project, as discussed in Chapter 10, contains a central focal point (its Software Development Library) for all project-generated materials. The programming data base being accumulated by an ongoing project is owned, administered, controlled, and maintained by the SDL for that project. However, the automated tools that are used by all projects to create, update, and extract information from the data base will not be considered here to be a part of any particular project SDL, but, rather, to be a part of the centralized programming support library within the standard software production system.

In actual developmental operations, a project may interface with its SDL totally through the secretariat, as in Chief Programmer Team [24] operations, or the team may interface with its SDL directly. Regardless of the interfacing mode, the contents of a project SDL are entirely determined

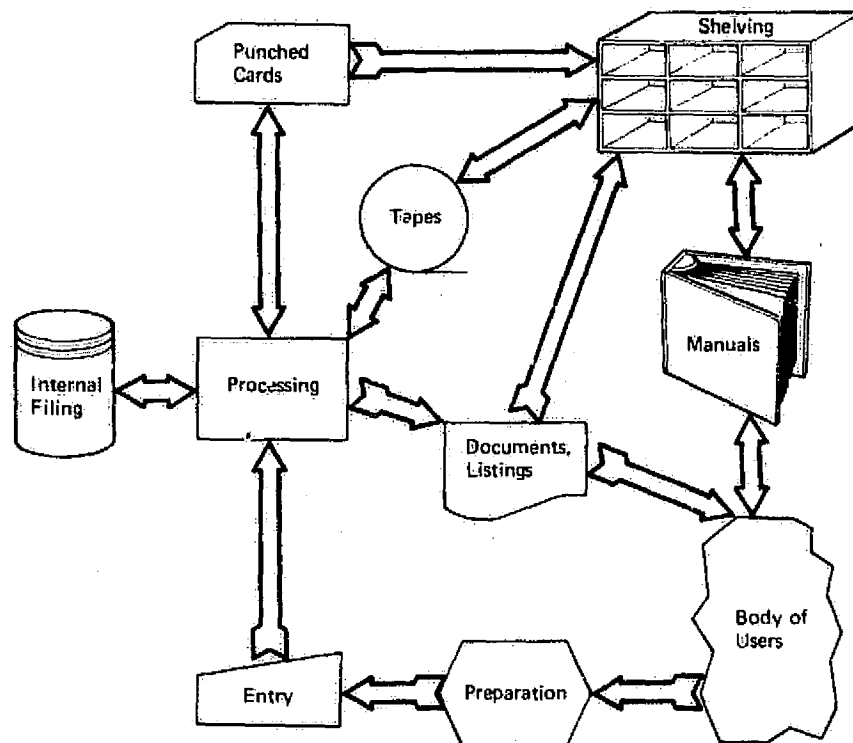


Figure 17-4. Secretariat services

by personnel within that development project, not by the secretariat. The format of a project SDL data base, however, is dictated by the standards of the production system, not the development team (Figure 17-5). The secretariat is responsible for maintaining control, integrity, and security of all software items placed within its purview.

Except for normal "housekeeping" operations, all clerical support tasks carried out by the secretariat are performed only on the direct request of project personnel, and then only when such support can be provided by the secretariat without project assistance or direct supervision. Directions from the project may, for example, come by way of original graphics or coding sheets for entry, by way of marked-up listings for update, or through any medium for which a standard secretariat support procedure exists.

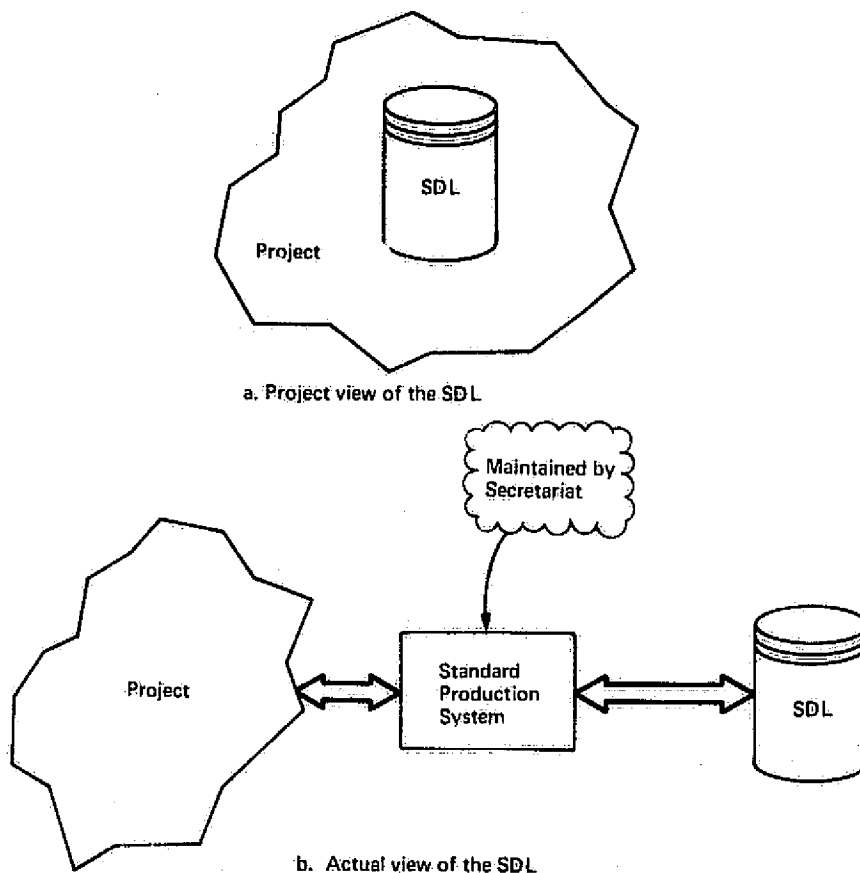


Figure 17-5. Virtual and actual views of the SDL (contents of the SDL are determined by the project, but the format of the data base is set by standards of the production system)

In operations where secretariat support is absent or limited, the standard support procedures and processors should be available for use directly by development team personnel. However, the advantage in cost and accuracy of having human clerical tasks performed by clerically trained individuals instead of programmers is obvious.

To support software developments, the secretariat needs a conglomerate set of skills that cut across clerical and technical boundaries. Personnel engaged in clerical support of programming require a set of qualifications such as typing, filing, and other business practices normal to an office environment; however, since the computer system forms the medium for these activities, support individuals must also be skilled in the use of keypunch or keyboard terminal equipment and must understand the procedures for preparing data and performing each intended task on the computer.

If the secretariat is also charged with the creation and sustaining tasks associated with the automated support functions, then it must contain personnel of the type and caliber found in the regular development projects and in the sustaining engineering and maintenance groups. If training of project personnel in the use of accepted organizational software development disciplines, organizational standards, human/machine interfaces, project/secretariat interfaces, etc., are also required, then personnel with skills appropriate to perform such training are needed. If enforcement of standards or software configuration integrity, control, and management are named as required development support functions, then appropriate personnel to administer such functions may also appear in the secretariat.

17.1.5 Host System and Environment Considerations

At this point, let me define what is probably a minimum set of requirements for a host system to accommodate the computer library and operations within the production system to be discussed. These requirements are not meant to specify a recommended set of equipment; the project size, equipment on hand, and intended tasks also affect the maximally effective set rather drastically. Instead, the configuration given here is meant to display my assumptions relative to the production system needs.

The host system envisioned is depicted schematically in Figure 17-6. The complement of equipment as seen by each user includes an interactive keyboard terminal and card reader for data and program entry, magnetic tapes and online direct access media for data storage, and a high-speed printer, character display terminal, and plotter for outputs.

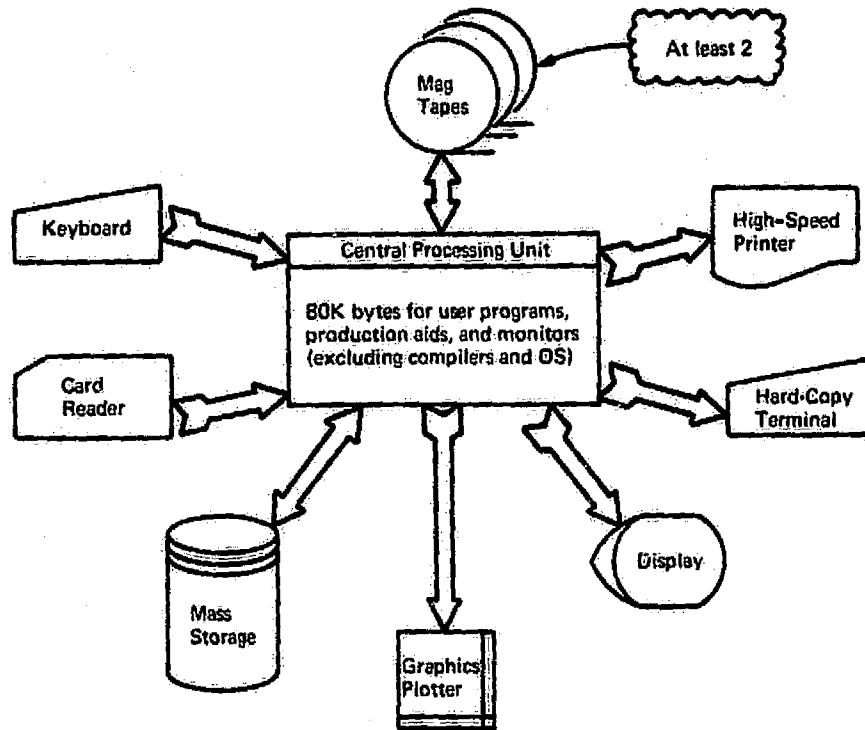


Figure 17-6. Standard host system hardware

Since much of every software development task is the development of easily readable documentation, I will suppose that I/O and storage media support both upper case and lower case alphabetic characters. A CPU size of 80K bytes over and above operating system and compiler overheads has been estimated necessary by Tinanoff and Luppino [25]; however, much can still be accomplished within host systems having smaller CPUs.

I will suppose that the host operating system, which controls the execution of the production system tasks (Figure 17-7), also controls the allocation of storage on the mass storage device. If more than one user is required to operate the production system at one time in concurrency, then the operating system will also administer and accommodate the allocation of resources among these users.

I concede that existing systems-level software will probably need to be revised, rewritten, or refined before it can be integrated into a uniform interface across the standard production system. Such utilities as file I/O handlers, program loaders, linkage editors, compilers, assemblers, text

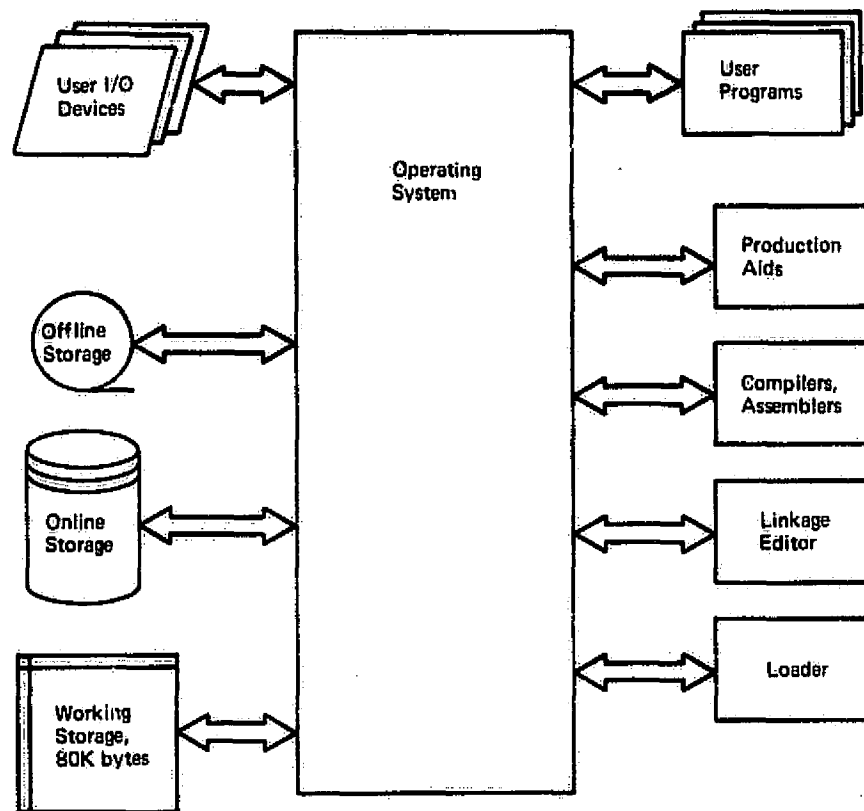


Figure 17-7. The standard production system interface configuration (the operating system administers execution of programs, allocates and manages storage and other resources)

editors, and other typical vendor-supplied software items can often be modified (especially on mini- and micro-computer hosts) to form the base for the standard software production system and thereby ease the development task that implements the standard system, at least on an interim basis. Substandard software should be replaced as soon as it is feasible to do so.

The use of a higher-level language (concurrent Pascal) has been demonstrated to be effective in writing operating system software [26] by Brinch-Hansen, who estimated a 20-30 man-year effort was accomplished by him in only a few months. From the production point of view, he concludes, it is both realistic and attractive to replace huge, ineffective "general purpose" operating systems with a range of small, efficient systems for special purposes.

17.1.6 User Considerations

One must recognize that the efficiency of a team producing software is influenced to a significant degree by the intended efficiency of the software to be produced and by the production tools available to do the job. For example, if a program must (unquestionably) be written in assembly language for a provably necessary enhanced execution speed, the productivity of the programmer is likely to be lower than if programming could take place using a higher-level language having a less speed-efficient compiler, unless the production system can supply services to equalize this imbalance. Brinch-Hansen [26] has estimated that only about 4% of an operating system needs to be in assembly code.

Another capability that influences development productivity is its operational mode. An interactive operational environment permits its users an immediate view of the response of a program, but the computer costs for these are usually somewhat greater than those for the same views supplied in a queued batch environment. One generally finds that there is an overall developmental cost savings when there is an interactive capability due to the decrease in user time required to interact with the system. Some inefficiency in computer operations can thus usually be tolerated (at least in non-real-time programs) whenever the rise in computer costs due to this inefficiency is dominated by personnel cost savings, brought about by improved usability.

Software development spans a time when a great deal of user interaction with the evolving program is required, regardless of whether the environment is interactive or not. Provisions to facilitate interactive interplay during this time are probably very advisable in terms of development cost savings, even if these raise running costs. However, if the operational usage of the program must thereafter continually pay for inefficiencies that facilitated program development, the cost savings may be eventually reversed.

To counteract such a possibility, one might propose a production system that would balance or equalize program life cycle costs by making the development task, perhaps, somewhat harder for the sake of decreased later operational costs. Alternatively, and more usefully, the system can contain modes and processors which promote both rapid development and efficient operations; this chapter supports the latter alternative.

The standard software production system of this chapter will thus support both interpretive simulation and compiled modes of operation. Programs can be developed interactively using the interpreter, or processor executing an intermediate code, for checkout and testing. Once correct, these programs may be compiled and optimized for acceptance testing,

delivery, and subsequent operations. The implications of this position on the development of programming languages for the system have been explored and endorsed by Wegbreit [27].

17.2 THE STANDARD PRODUCTION SYSTEM SUPPORT LIBRARY

As I indicated earlier, the support library contains not only the computer programs that aid software production, but, also, the manuals and practices that enable users to employ these aids. The aids themselves represent various mixes of functions that together provide the following categories of support:

- Word processing
- Graphics generation
- Management status monitoring and reporting
- Programming of routine or clerical tasks
- Program generation
- Program loading, linking, and execution
- Program analysis and performance monitoring
- Program repair
- Source data management
- Data protection and integrity

To varying extents, each of the aids will be characterized by such operational qualities as:

- Cost-effective operation
 - Reliability
 - Generality and power
 - Application independence
 - Simplicity of use
 - Conciseness and terseness in user interplay
 - Naturalness of interplay dialogue
 - Consistency with other aids
 - System-independent operation
- C-3

Cost effectiveness, of course, relates to the cost of performing a task using an aid, as opposed to performing the task without it; reliability refers to the average degree to which the aid performs effectively in program production, as compared to its advertized full capability. Generality, power, and application independence all refer to the scope of tasks accomplished by the aid: to the number of different tasks performed, to the amount of work accomplished by each task, and to the range of projects that may find the aid useful.

Simplicity in use characterizes features that eliminate unnecessary variations in notations; such features enhancing simplicity make an aid easier to use because they reduce the number of forms and concepts that must be learned. Conciseness and terseness of expression enhance usability by reducing the user/machine interface dialogue. Naturalness is a measure of the degree to which the symbology of user/machine communication is humanly understandable and easily remembered over long periods of time.

Consistency with other aids refers to an overall conformance with usage standards, coordinated with respect to redundancies among the various aids. Consistency would, for example, require that whenever two aids perform the same functions (overlap in capabilities), then the input forms and protocols would be the same for each. Identical forms would be required, whether input emanates from a terminal, file, or card reader; output forms, ditto.

More specifically, suppose that aids A and B both can operate on files of data, and each can duplicate, and thereby create new files; consistency would not, then, permit using differing syntaxes, such as:

COPY *file1* TO *file2*

DUPLICATE *file2* FROM *file1*

but would require conformance to a common syntax, say, the first. No third aid, say, C, could use this syntax to rename *file1* as *file2* but, instead, would have some different syntax, as:

MOVE *file1* TO *file2*

with a natural "semantic proximity." Finally, no fourth aid, D, could then use the MOVE syntax to deposit values into variable storage, but, instead, would have to employ a syntax with a wider implicit "semantic differential," such as:

LET *variable* = *expression*

System-independent operation refers to constancy in operation as viewed by the user, and has been discussed earlier (Section 17.1.5).

I will not attempt in these pages to provide a complete or detailed set of requirements or functional specifications for systematizing and automating a programming data base. Rather, I will try to identify and outline the salient concepts, key features, skeletal format, and rationale for a typical programming system support library. A more detailed set of typical requirements and specifications for a "Program Support Library" for use by Chief Programmer Teams [24] has been defined in the works of Luppino and others [25,28].

Figure 17-8 depicts the skeletal structure of the programming data base and the processors forming the basis for the standard production support library to be discussed. The processors to be discussed are those for languages, automatic flowcharting, text processing, and management reporting. Linkage editors, loaders, job-control supervisors, etc., will not be covered. As a result, the design, documentation, program, and management data base entries are perhaps more fully described than are some of the other files and documents in the data base.

17.3 STANDARD PROGRAMMING LANGUAGES AND LANGUAGE STANDARDS

Programming languages form the primary tools by which computer programs are constructed. Without a language well-suited to the task at hand, the most elegant of secondary tools will prove inadequate for composing high-quality programs, albeit secondary tools may permit one to build the poor program more quickly or better than would take place without such aids.

Those features of a standard language that implement procedural hierarchic refinement and control logic structures constitute what I will refer to as the *control sublanguage* of the language. Much of this text has already been devoted to discussion of topics that influence the design of a control sublanguage; Chapter 7 and Appendix G contain standards for the control sublanguage (CRISP) that I shall be using here in examples.

Those features of a standard language that are not part of the control sublanguage and that are not concerned with I/O constitute what I shall refer to as the *base sublanguage*.

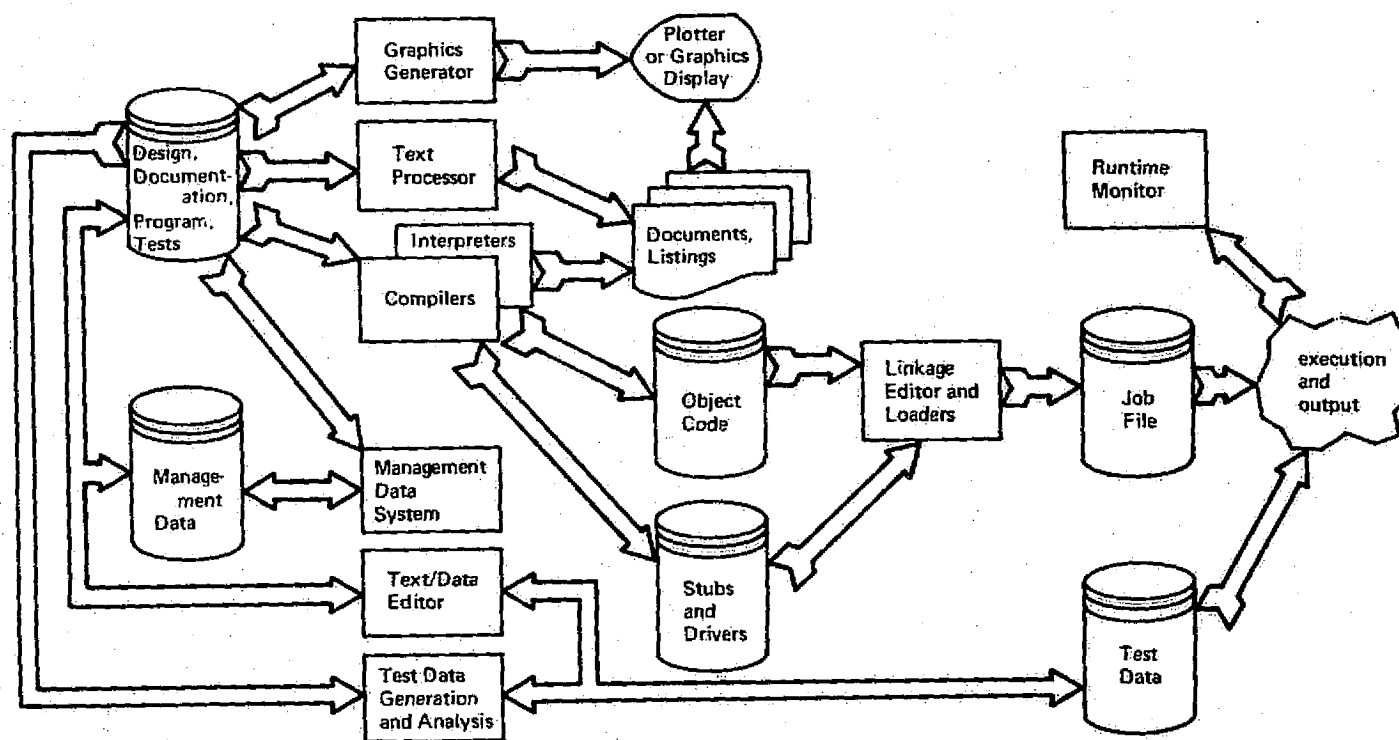


Figure 17-8. Skeletal structure of the Standard Production System Support Facility

The base language deals principally with data declaration and storage management, scalar types, features for data structuring, and operations on data. The Department of Defense requirements document [29] for higher-order computer languages contains an excellent discussion of features needed in a modern base sublanguage.

The remainder of the standard language is the *I/O sublanguage*. An excellent summary of I/O sublanguage capabilities for a number of higher-order language appears in the work of Goodenough *et al.* [30]. Further discussion of I/O capabilities appears in Section 17.3.4, following.

It would be preferable, in the interests of standardization and language commonality, to have only one standard language in the library repertoire. There is ample evidence [31,32] that such a requirement, however, is not a pragmatic one when implementation and operational costs are considered. As an alternative, then, there may be a set of standard languages, each coordinated with the others, all available to users, and all rigorously policed and maintained.

A coordinated set of standard languages should then differ only in the ranges of each of its component sublanguages, appropriate to the intended application set (non-real-time languages, for example, would not contain the FORK-JOIN structure); the system programming language may permit certain bit-level manipulations; etc. Features common to any two programming languages must, for standardization, have the same syntax.

It is imperative that each implementation not contain source-language features which are not defined in the language specification, and, moreover, any implementation of a language feature not explicitly permitted by the language must be expressly forbidden. This is necessary to guarantee that the use of programs and software subsystems will not be restricted to a particular host by virtue of its having a unique version of the language.

Such a restriction represents a commitment to freeze each source language, inhibit innovations and growth, and confine the language set to the current state of the art. In return, standardization buys stability, wider applicability of software tools, reusable software, greater software visibility, and increased payoff for tool-building efforts. Standardization does not, however, necessarily disallow optimizations which are host-system unique, nor does it prevent growth and modification as authorized by a responsible change control board.

It is equally important that every implementation of a particular language implements the entire language. If individual processors implement only a subset of a language, there is then no chance for software portability, and there is no guarantee that users may access standard

supported libraries or application programs implemented using another version of the same language.

Requiring that the full language be implemented will be expensive only if the language is large, complex, and non-uniform. A number of smaller compatible languages that can each communicate with the other and with libraries of specialized features, support packages, and complex operations should be the goal.

There is no room in this text to discuss all the factors to be taken into account in language design. The interested reader is referred to the interesting works of Nicholls [33] and others [29,30,31,32] for such information. A few highlights are, however, worth mention here.

In keeping with user utility, those tasks of a routine nature associated with language processing should be included and standard for all languages in the standard set. These include (1) automatic type checking of variables, functions, and parameters; (2) runtime checking of data-range integrity (e.g., subscripts within range); (3) automatic formatting and annotation of listings; (4) cross-referencing and indexing of variables, procedure names, literals, and other useful program elements; (5) path-analysis, program tracing, and other debugging features under programmer control; (6) performance monitoring for calibration of execution efficiency; (7) generation of meaningful diagnostic messages for anomalous behavior; and (8) linkage generation to independent compilation units, with consistency checks on passed parameters.

17.3.1 Structuring Features

Without discussing the detailed syntax and problem-oriented performance features of a programming language, let me address some of the structural and methodology-oriented aspects of the compiler which implements that language. (I am excluding the assembly language processor as a "compiler" here.) I envision that compiler as forming a concrete medium for the program procedural and data design (embodying a program design language, such as CRISP-PDL), and run the gamut from there all the way down, in hierarchically refined stages, to the executable code.

At the highest level, the compiler acts as a CRISP-PDL processor, accumulating program design algorithms, cataloging procedures, cross-referencing names, and matching (and linking) such things as procedural name usages with procedure definitions. Those procedures, functions, operators, data names, etc., which appear referenced, but which have not yet been given formal descriptions in the design and are not recognizable as elements of the programming language, are used to inform the programmers that such items need refinement. Eventually, when the design

is complete and when all refinements have reached the programming language level, then no unresolved references appear, and the output code is the completed program.

The source descriptions in the early design can be viewed as "non-compileable," as no compiler output need appear unless desired. However, the compiler is continually prompting the programmer, "What do you mean by..." naming a textual string used by the programmer, but as yet unrecognized by the compiler. At each stage, the compiler compiles what it can, links what procedures and data names it can, and informs the user of items it cannot handle as yet; in addition, it flags syntactic errors for removal.

Such a compiler can even create dummy stubs automatically (or interactively), given an outline of what stubs in general should do (for example, do nothing, print the name invoking the stub and return, prompt the user for input, etc.). That is, the compiler is capable of producing an executable program in some form or other whenever directed, at any point in the program development.

The first code produced by the compiler is probably interpretive, and is, perhaps, incrementally compiled, procedure by procedure. The production mode is interactive, and the program status can be probed at any point where the execution pauses for programmer query. Values for variables can be viewed or altered during such pauses, and the program can then be continued until a later pause. Because it works in this incremental, interpretive, interactive mode, the compiler needs to have the capability to display and alter values by symbolic name, as well as to edit and to re-execute parts of the program. The final program, produced by the compiler, is an optimized executable code, suited to the operational mode desired (interactive or batch or both).

As for languages: Even with the capability for hierarchic, semantic refinement built into a compiler, it is a fundamental error to believe that program developers should be confined to thinking in a procedurally oriented language. There will always be a need for a "cogitation medium" separate from a computer. The major responsibility in language design rests upon the creation of a tool which facilitates stating the solution to a class of problems once the solver has reached that point in the solution process where computerized aid can feasibly be standardized. The compiler I have described above attempts to aid the solution process by bridging the gap between certain program abstractions and their concrete representations in code.

17.3.2 An Example of Hierarchic Compiling: Ordered Search

Before going into more detail concerning features, syntax, and semantics of a language suitable for the standard production system, let me illustrate, by means of a hypothetical compiler, how the procedural design abstraction hierarchy might be accommodated.

Problem: A set of integers is contained in the 1-dimensional array, TABLE of size N; the integers are ordered, so that $TABLE[i] \leq TABLE[i + 1]$, brackets denoting subscripts. Design and implement a procedure, SEARCH(VAL, FOUND), to seek the integer variable VAL among the TABLE items, and set a Boolean variable FOUND, to true or false accordingly.

Solution: The solution method* is to partition portions of the array iteratively for search into a left part and right part by some as-yet unspecified criterion. The first part of this solution is, then, the CRISP-PDL procedure:

```

procedure: SEARCH(VAL: integer, FOUND: boolean)
  Select initial portion to be searched
  loop while (more than one element in portion)
    reduce portion to be searched
  repeat
    examine chosen part and set FOUND
  endprocedure

```

At this point, the compiler has recognized only the underlined keywords, the procedure name and variables to be passed, and the procedure and loop structures. The compiler then asks for definitions of the strings, "Select initial portion to be searched," "more than one element in portion," "reduce portion to be searched," and "examine chosen part and set FOUND."

The programmer responds, in this hypothetical language, with a set of refinements:

```

to select initial portion to be searched
  declare L, R: integer local to SEARCH
  L:=1, R:=N
endto

```

* Per Brinch-Hansen, Structured Programming Class Notes, JPL Employee Development Class, Spring, 1975.

```

to reduce portion to be searched
  partition current portion into left and right parts
  if (solution impossible in left part)
    select right part
  (else)
    select left part
  endif
endto

```

The programmer has chosen not to respond at this time to the other elements asked for above. The compiler then asks for definition of elements "partition current portion into left and right parts," "solution impossible in left part," "select left part," and "select right part." The next set of refinements entered are:

```

to partition current portion into left and right parts
  set C to rightmost index of left part
endto

```

```

macro: solution impossible in left part means
  TABLE[C] < VAL
endmacro

```

```

to select left part
  R:=C
endto

```

```

to select right part
  L:=C+1
endto

```

```

macro: more than one element in portion means
  L<R
endmacro

```

```

to examine chosen part and set FOUND
  if (VAL=TABLE[L])
    FOUND:=TRUE
  (else)
    FOUND:=FALSE
  endif
endto

```

At this point, the compiler has recognized all elements of the solution except "set C to rightmost index of left part," and it finds no declarations for ARRAY and C, whereupon is entered:


```

declare C: integer local to SEARCH,
TABLE: array [1:N] of integer global,
N: integer global

```

Whether the search is a linear sequencing through TABLE, or a binary segmentation of the search portion into two nearly equal parts on each cycle, or almost any other kind of ordered search, it is centralized and controlled by the single procedure "set C to rightmost index of left part," although it is possible that some searches (e.g., a Fibonacci partitioning [34]) might also require some additions to "select initial portion to be searched."

Carrying the example here further, I shall suppose a binary search is in order, in which case the following refinement is entered so as to divide the search region by 2 in each cycle:

```

to set C to rightmost index of left part
  C := IP((L+R)/2) < *truncates to Integer Part* >
endto

```

The concrete declarations and operations above, such as "declare..." and "C := IP((L+R)/2)," are assumed to be in that regular part of the language syntax not needing further refinement. I have taken liberty in assuming that the reader comprehends what is meant by these constructions in this hypothetical language without explicit definition (which, of course, the hypothetical user's manual would provide).

The program development is now complete, as all elements have been recognized by the compiler. On seeing how procedures are defined, the compiler then provides a suitable linking method. In the example shown, all procedures are probably inserted in-line in the object code.

17.3.3 Data Structuring Features

Each programming language in the system must, of necessity, be able to declare data structures and to operate on those structures as appropriate to solve the problem at hand, and each of these languages, therefore, needs a common, standard syntax with which to declare its data. A programming language tends automatically to set standards for data structure declarations, and the standard syntax should, therefore, be designed so as to permit hierarchic development of data structure specifications as discussed earlier in Chapters 4 and 12.

In Chapter 2, I defined a data structure as a representation of the ordering and accessibility relationships among data items, without regard to storage structures, or implementation considerations. In this sense, data

structures lie midway in the mapping process that transforms the problem-solution domain into the storage and computation domain. Data structures are characterized by a set of *attributes* accorded to the data items to be manipulated. Among these attributes are:

- *Ranges of values for data items*
- *Relationships among data items*
- *Means of declaration and creation*
- *Rights and means of access*
- *Valid sets of operations when accessed*

The specification of a set of attributes defines a *data type*. The simplest data types for data items in a language are called *fundamental scalar types*. In FORTRAN, for example, integers and reals are the primary fundamental scalar types permitted. *Extended scalar types* refer to wider classes of scalars defined in or through the language syntax, but without direct accessibility to the underlying scalars that form the extended types. For example, a FORTRAN complex number is formed from two reals (simple scalars), and all references to these components are explicit, being achieved only through the defined set of operations and functions for complex numbers.

Data that contain scalar types as elements or sub-elements are called *structured types*. The simplest structural types in a language are said to be *fundamental* or "built-in" structural types. FORTRAN permits only fixed-dimensional scalar arrays as its fundamental structure. PASCAL contains, in addition, record and file structures.

Declaration is the notification to the compiler of data type attributes; when storage is actually allocated at runtime or reserved statically at compile-time for later operations, the structure is said to have been *created* or *bound*. The *access domain* of a structure is composed of that set of operations permitted to read, write, or otherwise manipulate it.

For reliability, most modern languages now provide some measures of protection and privacy so as to catch the simpler forms of inadvertent violation of access domains. Such languages are said to be *typed*, as opposed to languages that permit arbitrary operations on a variable. For example, FORTRAN, which permits character strings to be stored in integer-declared arrays and operated on either as integers or characters, is untyped in this respect. PASCAL, on the other hand, is fully typed, not even allowing integers to be added to reals without explicit type conversion.

Access attributes are declared forms of *access rights*, such as *literal* or *constant* (read only) and *variable* (read/write), qualified by such modifiers as *global* (accessible by the entire program or a set of programs), *local to...* (accessible only within a given list of modules and their submodules), *readable by...* (read-access only by listed modules and their submodules), and *writable by...* (write-access only by listed modules and their submodules).

Data structures are also characterizable by storage creation attributes, such as *static* allocation (fixed at compile-time), *controlled* (determined dynamically at runtime), *automatic* (dynamic runtime allocation of temporary data structures and release after use), and variants of these, such as *based* storage (controlled storage in which temporary copies are created, stacked, manipulated, and destroyed in block-structured programs).

Data structure declarations may also establish *initial values* to be present upon creation.

An *abstract data type* is that which can be described by a cascading hierarchy of data-type definitions whose primitives are fundamental types in the programming language. The design procedures outlined in Chapters 4 and 12, in fact, describe data structure design as the process of abstract data type definition.

Programming languages that provide abstract data type declaration facilities to varying degrees have been in existence since about 1967 when SIMULA-67 appeared. Since then, there have been an ever-increasing number of such languages, e.g., ALGOL-68, PASCAL, concurrent PASCAL, and PASQUAL. A useful bibliography of papers concerning data types and programming languages may be found in the work of Tennent [35]. Specifications for a Data Description Language (DDL) have also been published [31] by the CODASYL group. The Department of Defense requirements for its higher-order language [29] state that it be typed, allow for the definition of new data types and operations within programs, and permit ranges of values that can be associated with a variable, array, or record component to be any built-in type, any defined type, or a contiguous subsequence of any enumeration type.

Specification techniques for data abstractions have been treated by Liskov and Zilles [36], who concluded that there is no single specification technique that is universally better than others in terms of providing mathematical inambiguity, ease of construction, ease of comprehension, and minimality of expression in describing the pertinent usage attributes of the structure, without having to divulge purely constructional details extraneous to the problem.

Hoare [37], in fact, concluded that it is totally unrealistic to suppose that any high-level language or automatic process (or even a generalized, integrated data-base management system) could be made to produce satisfactory implementations from abstract definitions, in the general case. Rather, it always will be necessary for the programmer to synthesize some of his abstractions using fundamental operations and declarations, programmed levels of access, coding conventions and standards, and documented data relationships.

The syntax of the base language must, therefore, accommodate each of the attributes necessary for humans to build concrete data structures from virtual abstract data types. The basic attributes needed include an adequate set of fundamental scalar and structure types from which other types can be generated. Such a set might well include the scalar types

- Integer; bounded integer
- Real; bounded real
- Character; string of characters
- Boolean; logical
- Complex
- Member of enumerated set
- Double precision (integer, real, complex)

and the fundamental structures over each scalar type may well include

- Array (indexed access, single scalar type elements)
- Record (field-name access, arbitrary scalar type elements)
- Buffer, or sequence (sequential access, single scalar type elements)
- Set (membership access, discrete scalar type elements)
- List (chained access via pointers to single scalar type or record elements)
- String of boolean

In addition, the following first-order compound-structural types contribute enormously toward the ease by which data abstractions can be encoded:

- Array of records
- File of records
- List of records

In each of the compound structured types above, the records have scalar elements only.

The capability to compound structural types arbitrarily to the cases in which elements identified above as scalars can instead be any previously specified data type is precisely what is needed for abstract data type definitions. The extent to which such features can be made feasible within a programming language syntax determines how much of the abstraction must be programmed using other means.

In order that storage binding time be programmable, the base language must not only permit static allocations of data structures, but, also, dynamic allocations and reallocations during execution. Automatic and based storage can generally be simulated by special attention to controlled storage procedures. The capability to declare an *initial value* to be supplied when the structure is created may be included as a base-language requirement.

In order that storage locations be bound in a conveniently programmable way, data ownership declaration options are needed. Structures occupying a global area should link to access functions by the same means used to declare those structures; for example, a named variable declared to be global should be accessible by its name throughout the program, regardless of the order in which variables are declared in its separately compiled parts. This, of course, is contrary to the FORTRAN practice of locating elements in COMMON by position, rather than by name. The base-language processor may require a special linking loader to accommodate separately compiled modules.

Linking of structures declared to be scoped within a module and accessible by any of its descendant (perhaps separately compiled) submodules may also require special features within the compiler and loader not only to communicate structure names but, perhaps, the module hierarchy as well. The limited use of data structures by a set of operations is a subcase of a more general problem that in which access rights to resources are controlled and administered by the language processor. Certain structures (literals) are meant to be read-only by any module; others (variable) are meant to be arbitrarily read or written into by any authorized module; still others conceivably may be read-only to some modules and write-only (or read/write) to others.

A language that checks and administers such rights of access to resources must implicitly or explicitly permit programmers to communicate these rights to the structures being accessed. Most commonly, localization of access only is available, achieved through separate compilation or through block structure in the language; functions and subroutines, which are externally referenceable but compiled together using mutual data among themselves, then form a level of access to their local data structures.

The standard languages should, however, support the definition of levels of access to resources within each compilation, and not necessitate the, perhaps, unnatural dislocation of code segments into separate compile modules. Moreover, localization of functional access within levels should be possible; access functions that form a sublevel of access within a given level of access should not be referenceable above that level of access.

17.3.4 Input/Output Capabilities

The processes of input and output are in truth merely mappings from data structures in one medium to data structures in another. Among data structures in core, such mappings are frequently referred to as *type conversions*; however, in I/O, the mappings are generally called *formatting*. An *I/O type* consists of a set of objects or values associated with physical devices characterized by special selection and assignment operators, special organizations and associated sets of operations, and an end-of-I/O property. Most higher-order languages support a variety of different I/O devices, transmission modes, access rules, organizational protocols, and structural attributes.

I/O data types are not always explicit in a language, but a language that supports I/O functions must include, in the narrowest sense, the following operations and exception conditions:

- *Open* medium for access, assign buffers (can be implicit)
- *Close* medium access, deallocate buffers (may not be necessary)
- *Read*, or select an object from the medium (buffer)
- *Write*, or assign an object into a medium (buffer)
- *End-of-medium*, no more data accessible from medium
- *No-such-device*, medium cannot be identified
- *Access-error*, improper access operation
- *Physical-error*, medium physical failure

An open I/O medium has at least a certain minimum set of assigned attributes, among which are

- Medium type
- Label or name (if a file)
- Access restrictions (read-only, write-only, etc.)
- Buffer designation

Additional attributes define the I/O data structure for proper conversion into or from physical storage (character vs. binary, record vs. stream access,

stream in character mode vs. line mode, records blocked vs. unblocked, formatted vs. free-form, etc.).

The interfaces (Figure 17-9) between the base sublanguage and the I/O sublanguage are the I/O buffer, format specification, and attribute table. The buffer must be capable of receiving or being assigned any valid data type addressable in the base language (actually, just scalar data types and combinations of them that can be packed into aggregates with word or character alignment). Both formal and default formats must be accommodated, and format generation must be under programmer control.

The programmability of the I/O sublanguage is a matter of assigning or accessing buffer values, setting attribute table entries, specifying format

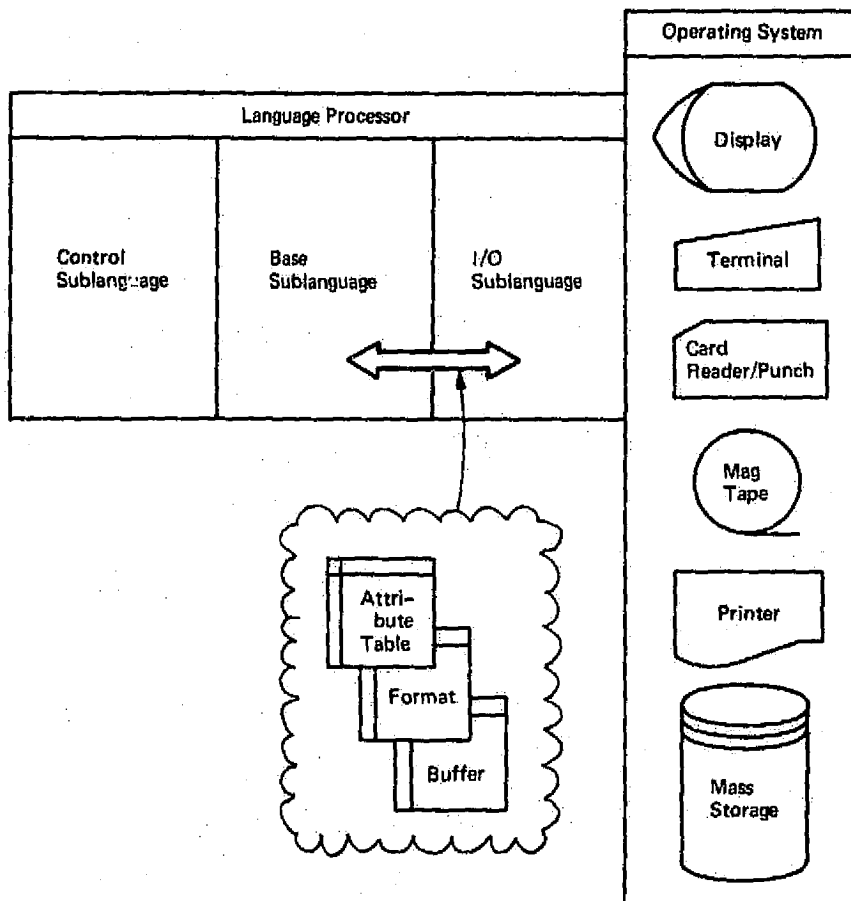


Figure 17-9. Interfaces between control, base, and I/O sublanguages

parameters, and reacting to exception conditions. The ability to support I/O functions to special devices is partially dependent, however, on the presence of data types that map directly to the values implicit in the devices. In the interests of minimizing host system dependencies and promoting language commonality among installations, it is important that the allowable set of I/O functions be limited to a subset of capabilities supportable on all standard hosts.

For this reason, the internal attributes of the I/O media should not be accessible by the applications programmer. Formats and other programmable attributes should probably be recordable directly on output media, right along with the output data, so that automatic type checking can be accommodated (to some limited degree) when that data is read back in by a program. Devices (such as keyboard terminals) that are input-only must then handle only a single type (e.g., character), but with conversions to scalar types and aggregation into structures via a specified input format.

17.4 CRISP-PDL PROCESSING

The quality of a computer program can often be significantly related to the design medium in which that program is developed, embryonically and evolutionarily, from the ideas that permeate the programmer's mind, to the completed programming specification. Such a medium must foster the expression of such ideas easily and quickly (sometimes before they fade from memory), and must permit flexible and facile alterations, additions, and deletions to these ideas as the design evolves. Moreover, the expression of the design should be as graphic as a "picture of the program"—yet not be the program, nor constrained by the rigorous syntax of a compilable computer language. At the final evolutionary stage, such descriptions should form the principal program design document.

A "Program Design Language," or PDL,* is a formalized embodiment of such a design aid, and can take many forms. Probably the most familiar form of a PDL is flowcharting. Flowcharts have many advantages and many disadvantages, which will not be named or argued here, except to say that, while being very graphic, they are nevertheless very limited in what they can do for the program designer because of the limited space available for expression, because they are primarily procedure-oriented, because of their inflexibility toward alteration or revision, and because of the expense required to draft them in a "finished" form.

*This acronym also stands for "Procedure Definition Language," "Procedure Design Language," and other equivalents.

The compiler described earlier in Section 17.3.1 had the PDL capability built in, interactively building up the program through stepwise-refinement-dialog with the user. However, there is more to a programming task than can be expressed in a procedure-oriented language. There must also be a way to explain to readers of the documentation the significance of certain operations, the goals to be met by programmed functions before they are programmed, the assumptions that have been made at a particular point in the design, the reasons why the given constructions were used (or why others were ruled out), and illustrations which facilitate the understanding process.

Such program rationale can, of course, be kept apart from the program or merged into it somewhat in the form of comments; both methods have many admirable qualities: Comments tend to explain local effects in procedures at their points of occurrence. Separate narrative and graphics are usually better for analyses and more global explanations of program function.

In the remainder of this section, permit me to focus on some text-formatting and design-reporting features of a simple hypothetical CRISP-PDL processor consistent with later flowcharting requirements (Section 17.5). A more detailed description of CRISP-PDL may be found in Appendix G.

Each CRISP-PDL input line consists of possibly three fields: a prefix, a cosmetic, and the program text. The prefix contains possibly a step number within the module (usually chosen to correspond to a box on a flowchart) and, perhaps, a subroutine or function cross-reference. The cosmetic portion consists of spaces and vestigial flowlines, so as to present an indented listing, which displays many of the features of a flowchart as illustrated earlier in Chapter 7.

The text field of the input is of two varieties: a control-logic text and non-control-logic text. Control-logic text fields begin with a control keyword, such as if, loop, repeat, or a left parenthesis "(". Such keywords signal the processor to increase or decrease the indenting level and to add, delete, or modify the vestigial flowlines. The headings of nested structures (e.g., if, loop, else, etc.) increase the indenting level and add flowlines; endings of nested structures (e.g., endif, endcases, repeat, etc.) decrease the indenting level and eradicate flowlines. Abnormal and paranormal exits (exit, return, stop, and system) cause no change in indenting level, but do show a flowline exit of the current nesting level back to the appropriate level.

The output of the CRISP-PDL processor consists of a table of contents or module directory, a tier chart, and the cosmetized CRISP-PDL with cross-references of identifiers, subroutines, and functions. Figure 17-10 summarizes the CRISP-PDL inputs, processing functions, and outputs. Inputs are the source file being processed and control data that selects output options. Processing consists of cosmetizing source lines, as described earlier, the accumulation of module and identifier names, page numbers, and cross-reference material, and the output of such material as directed by control data.

A more detailed data-flow diagram is shown in Figure 17-11 for purposes of describing the functions more clearly. Neither the names of symbols nor the physical structure of the program internal data flows necessarily corresponds to those in the program internal design. Symbols are numbered to identify input (1), processing (2), or output (3), just as in Figure 17-10; an additional Dewey-decimal identifier serves to individualize the boxes into separate functions.

Text appearing between a module ender and the next module header is copied directly to the listing without cosmetization or diagnostic checking. Each line is scanned, however, since it potentially contains identifiers. It is just as important to locate identifiers in commentary as it is to locate them in the procedural listing.

Each module begins a new page, as does any narrative that follows a module end and a new module header. (Each module end signals a page advance.) The output report starts the CRISP-PDL listing at page 1 (the table of contents and front matter are given Roman numeral page numbers).

The output report, in its fullest form, consists of the following sections or parts, in order:

- Title page
- Table of contents
- Program directory

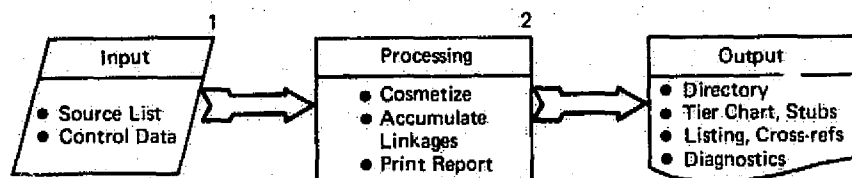


Figure 17-10. The CRISP-PDL HIPO diagram

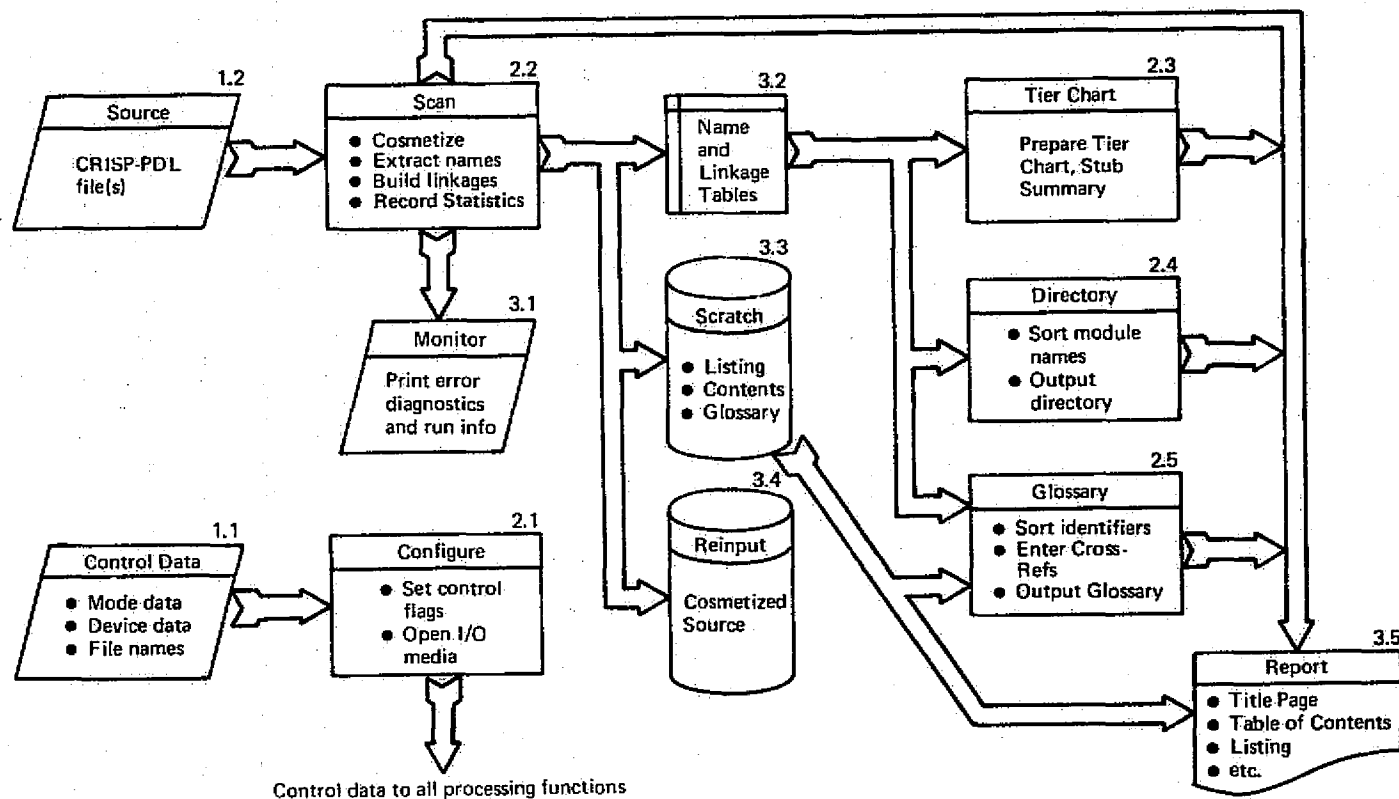


Figure 17-11. Conceptual data flow for purpose of describing CRISP-PDL functional behavior

- Tier chart
- Stub status report
- Cosmetized source listing, first module
- Intermodule text
- Cosmetized source listing, second module
- ...
- ...
- Glossary and cross-references
- Statistics of program

The full set of CRISP-PDL forms appears in Appendix G.

17.5 FLOWCHARTING FROM CRISP-PDL

The equivalence between CRISP-PDL and flowchart structures is not coincidental. The exact one-for-one correspondence was, by design, intended to facilitate software production by making design-level graphics available and suitable for program documentation. Programmers who claim they do not need flowcharts to design and implement programs can use CRISP-PDL as a much more writable, flexible, and (to them) readable specification of the program procedure. Processing the CRISP-PDL produces the more graphic documentation for others to read, automatically and at low cost.

Still others will, perhaps, choose to work with flowcharts on a primary basis, but using the entry and update capabilities of a text editor, rather than manually drawing and redrawing their own charts.

In this section I describe a simple flowcharter for the CRISP language, called CRISPFLOW. It is simpler than most general-purpose flowcharters, such as AUTOFLOW [38], BELLFLOW [39], etc., because it only draws structured flowcharts, one module to a page, and permits no off-page connectors; it conforms such charts to the layout standards used throughout the two volumes of this work.

The flowchart symbols plotted by CRISPFLOW will have varying dimensions according to the text to be contained in each. Nevertheless, standard ANSI [40] aspect ratios (width:height) are maintained. The letters "T" and "F" always label binary decision symbols, true always on the left. Structures always have straight-down flow paths; that is, branch-collecting

nodes are located exactly below the branching vertex, etc. CASE-clause identifiers become labels on the various paths emanating from the multi-decision branch. Figures 17-12 and 17-13 show an example of CRISP-FLOW input and output.

Horizontally striped symbols result upon encountering DO or CALL statements in the CRISP-PDL source; vertically striped symbols are drawn in response to the keyword CALLX. The remainder of the text on a line enters the box, the procedure name above and the comment field below.

Loop-collecting nodes are distinguished as open circles; decision-collecting nodes by filled-in circles.

CRISPFLOW does not chart off-page connections nor any other unstructured form. No matter how large a CRISP-PDL module is, it is sized to fit on a standard 21-1/2 x 28 cm (8-1/2 x 11 in.) page by scaling

```

PROCEDURE: SAMPLE <*18 SEPT 75*>                                MOD# 1.3.5

<*This sample module demonstrates the CRISPFLOW
<*syntax with a hypothetical message transmission
<*system. Statements denoted ST1 through ST4 are
<*unspecified here, and the READ routine is
<*external to the set of documentation for which
<*the flowchart is being produced.

.1      IF (UNALLOCATED)
.2      :   CASE (MODE)
.3      :   :   :->(1)
.4      :   :   :   ST1
.5      :   :   :   :->(2)
.6      :   :   :   ST2
.7/S1   :   :   :   ST3
.8      :   :   :   :->(3)
.9/XS2  :   :   :   ST4
.10     :   :   :   ...ENDCASES
.11     :   :   CALL OPEN (MODE)<*CHANNEL IS MODE NUMBER.*>
.12     :   :   :->(ELSE)
.13     :   :   LOOP WHILE (AVAILABLE)
.14     :   :   :   CALLX READ(CHR)<*READ CHARACTER*>
.15     :   :   :   CALL WRITE(CHR)<*WRITE CHARACTER*>
.16     :   :   :   REPEAT
.17     :   :   ...ENDIF
.18     DO CLOSE<*MESSAGE SENT*>
.19     DO RELEASE<*DISCONNECT CHANNEL*>
.20     ENDPROCEDURE

```

Figure 17-12. Example of CRISPFLOW source format for a PROCEDURE flowchart.

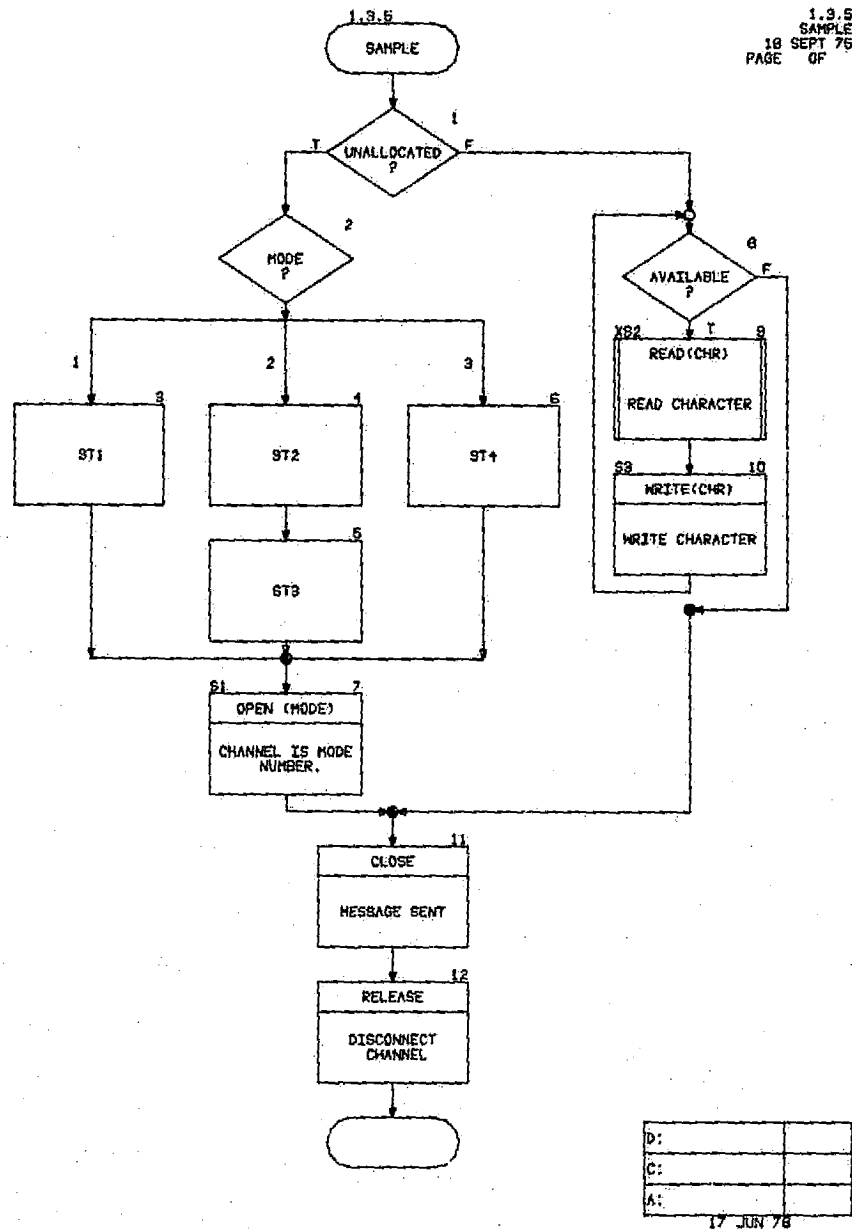


Figure 17-13. Output of CRISPFLOW processor for the SAMPLE procedure

down all symbols accordingly. The number of symbols that can be flowcharted is, therefore, set by readability, not the caprice of CRISP-FLOW.

The charting algorithm is very simple in concept, and worth mentioning here as an interesting design example. For each module in the CRISP-PDL source, the top-level control structure appears as the following three steps:

```
DO PARSE <*form a tree structure for the module*>
DO LAYOUT <*locate and size the chart on a page*>
DO DRAW <*plot symbols and flowlines on the page*>
```

The PARSE step scans each CRISP-PDL statement to extract the step number, cross-reference, keyword, and comment fields, storing this information into a node of a tree structure, representing the graph. This node is attached to the tree by arrangement of son/brother pointers associated with the nodes, to record the nesting hierarchy of statements in the CRISP-PDL. Figure 17-14 illustrates the tree format for the SAMPLE program.

The LAYOUT step applies a "super-box" approach to size and arrange the boxes on the page. The layout procedure scans the parser-generated tree in a post-order traverse (see Section 7.3.2 and [34]), and for each node forms a transparent "super-box," which encloses the super-boxes of all its subtrees (nested elements). The size of each super-box (stored with the other node information) is determined by the type of node (i.e., statement) to which it corresponds, together with the sizes of the super-boxes of each of its subtrees. The post-order walk permits the structure of the tree to be utilized as a "reduction system" or means to layout the chart in a comprehensive, bottom-up first pass through the tree. As the nodes at each level in the tree are scanned and "reduced" to super-boxes, they are used in turn to construct the super-boxes of the nodes at the next higher level. This procedure continues until the top-level statements of a module are combined into the super-box for the entire module.

The DRAW step, now having the size of the total chart, scales its super-box to fit the plot page, centers, and performs a pre-order scan [34] of the tree, during which it establishes coordinates, draws the boxes, fills them with text, and connects them with flowlines, arrows, and collecting nodes, all in one final pass through the tree.

The box-arrangement algorithm described above is a compromise between a "dynamic programming" approach and one that forestalls repetitive, brute-force search schemes, to allow rapid chart layout with only modest memory demands. The "super-box" approach yields very well

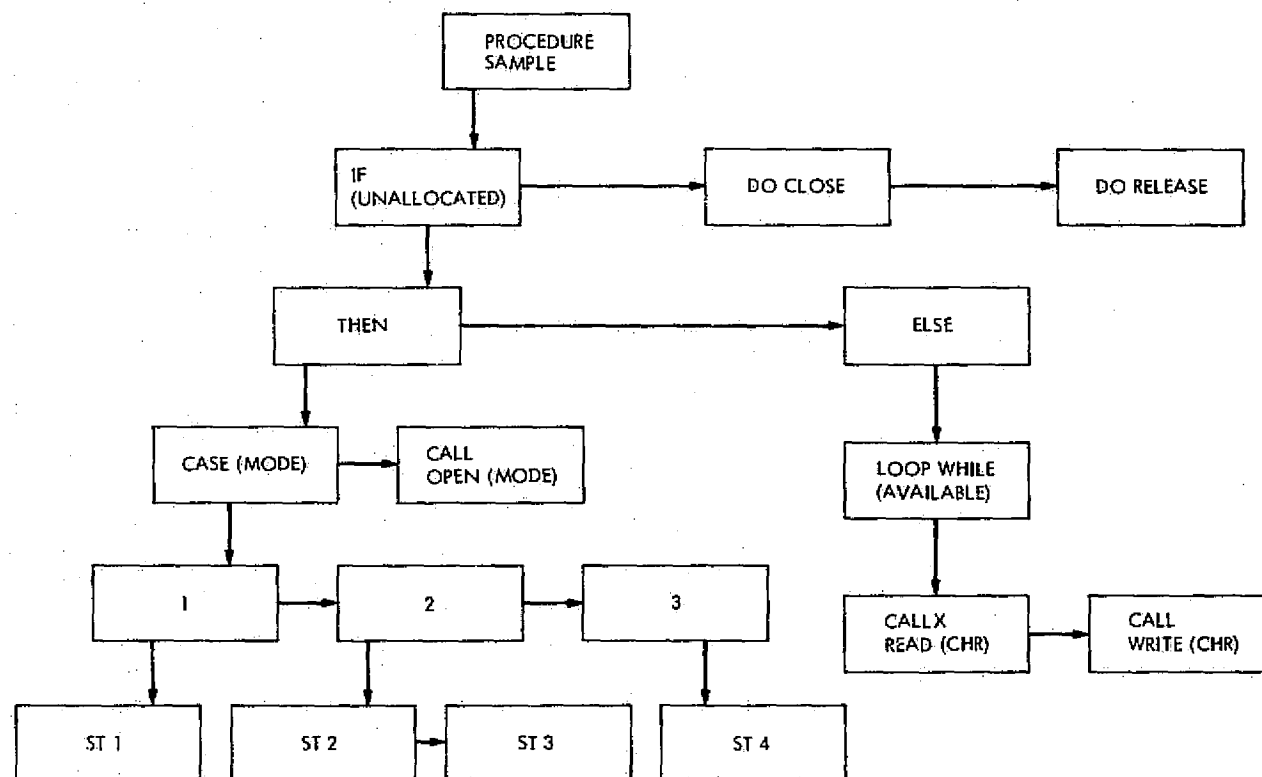


Figure 17-14. The binary parse tree of procedure "SAMPLE" (SON pointers exit the bottoms of boxes; BROTHER pointers leave the sides)

arranged charts without packing boxes as densely (or as "cleverly") as might be achievable using one of the slower generation schemes.

17.6 TEXT AND PROGRAM FILE EDITING

A means for text editing is one of the primary needs in every software development. Both the design documentation (procedure and narrative) and the source code (including data) for the compiler are in the form of text files that require constant application of the editor from inception to program delivery, and on into sustaining and maintenance throughout the program life cycle.

The text editor is, therefore, a general-purpose tool for program production whose syntax and semantics must be engineered to fit in uniformly with the other aids in the programming system. It is the means by which all files operating within the standard program production system are created and updated.

As with other aids mentioned in this chapter, a detailed set of requirements for a text editor is beyond the present scope. However, some of the highlights are worth discussing.

The first interesting aspect of the editor I would propose is a feature that, while administering changes to program elements (documentation, CRISP-PDL, code, tabular data, etc.), would at the same time perform some of the management functions needed to monitor program development. Some of the simple things that can be done are to (1) redate any module altered by the editor; (2) remove configuration control codes (in the comment field of a CRISP-PDL module header, for example) until the approval cycle (Section 10.4.2) reestablishes the module validity; (3) identify changes to either code, CRISP-PDL, or documentation which were not also accompanied by changes in the others as well; (4) automatically queue or identify all altered CRISP-PDL modules or narrative elements for flowcharting or other output; (5) automatically register program completion status reports with the management data base when modules are added or completed (or deleted); and (6) register statistics on editing transactions with the management data base. Such functions provide the visibility into programmer activities needed by management to assess the project's health, diagnose trouble spots, and monitor the team progress.

The other requirements on the text editor relate to its effectiveness as a productivity-increasing production tool. It should, of course, be able to operate on all text files used within the production system. For efficiency

and flexibility it should accommodate both sequential and random-access files, with unrestricted record and file lengths.

The editor needs to be both an interactive tool and a programmable one as well. That is, it should function directly in concert with the user at a terminal in interactive dialogue, and it should operate when given pre-stored or batch-entered instructions.

The programming language should be simple, easy to learn, and easy to remember, with syntax coordinated among the other languages in the standard set, as discussed earlier in this chapter. The user should be able to define (and modify) file and record structures, whether defined within program elements, or whether existing in some predefined "structure file."

Files may require both public/private and read/write security keys; the editor, moreover, should enjoy no special privilege with respect to private or read-only files. The user in each case is expected to supply proper access codes for the operations and files to be accessed. The philosophy for file security assumed here is one that protects users from non-malicious, inadvertent usages by others, rather than malicious breaches of security for illegal purposes. Protection of the latter type is needed in many applications, and can be added as a requirement in such cases (at some expense, no doubt).

Some of the features needed are:

- a. Editor can be initiated from another processor without loss of workspace or file status upon return to that processor. However, file editor does not access workspace of that other processor.
- b. Edits files even if already open in another processor. However, the file editor must provide for consistent usage (Chapter 6) of time-shared files.
- c. Edits records using syntax standard among similar line-editing capabilities in other processors.
- d. Extracts substrings from records by character position(s), free-form fields, user defined format, and field name in user specified record structure; it can use these in relational operations, in string expressions, in numeric conversion and arithmetic operations, and for generating file names.
- e. Inserts new records or deletes old ones at accessed location.
- f. Displays accessed record, extracted substrings, and computations on user terminal or on output files. Display can be conditioned on extracted or other computed data, and formatted as desired by user.

- g. Can copy files to tape, or tape to files; can name and rename files; can delete unwanted files; can concatenate files; can insert one file within another.

17.7 MANAGEMENT DATA AND STATUS REPORTING

Performance measurement in this section refers to management gauges of computer software production, rather than efficiency metrics on the software itself. However, the quality of the delivered product is certainly an important aspect of performance, and so the measurement of performance must not exclude quality as a performance factor. Therefore, in this context, *performance measurement may be reduced to the measurement of the capacity of a software development team to produce working, documented software, together with qualifying measurements of cost, time, and quality of product.* This function is closely coordinated with the ability of software management to estimate costs, manhours, and schedules, to assess manpower skills and required areas of specialization, and then to exercise measures of control over the emerging quality of product.

Basic to the establishment of such measurements is the establishment of standards, languages, and software production techniques, as discussed earlier in this work. Once this is done, a conceptual procedure for performance measurement is not difficult to hypothesize:

- a. List desirable performance qualities that are also measurable.
- b. Develop a mapping of performance units into a uniform scale of merit.
- c. Measure performance and normalize (i.e., convert to a unitless scale).
- d. Compute grade based on some formula (say, as a weighted sum of scores).

The point of view here is that science, technology, industry, and commerce are based on the ability to measure things or phenomena, describe the results in numerical terms, make comparisons, and make decisions based on these comparisons.

The hypothetical method given above is not without fault; associating value with measurement is often very subjective. But measured quantities need not be taken literally as absolutes in order to provide management with a useful tool. Indicators of trends may often be just as useful to a manager as knowing the causal relationship basing the measurements.

The ability to assess the performance of a software producer—individual or group—is basic to estimating costs and time and to exercising

corresponding control over the production process. At the same time, the ability to measure performance will not, of itself, provide good management technique. It will, however, permit responsible management to measure differences between alternatives and to react accordingly. The software manager assumed here will, therefore, be assumed to be already competent in his job, even without automated support; any automated capability he is provided should merely aid him in performing that job.

Every manager has an established need to collect and manipulate data directly concerned with performance and to use these data, or relationships among data items, in the performance of his job. If collected into a data base that can be accessed and queried by a computer, then there needs to be a well-defined, effective means by which its users perceive and interact with that data base. The form of this interface between the data base and the manager is, thus, particularly vital to the utility of that system.

17.7.1 The Management Forms Approach

The data-base/end-user-facility interface recommended by the CODASYL End User Facility Task Group [41] is one based on the "forms approach" to data base operations. The essence of this approach is that it enables the manager, among others, to think of and view the content of the software management data base as a series of two dimensional forms contained in files and folders (Figure 17-15).

There are three subclasses of forms that may be identified: perception forms, user forms, and worksheets (Figure 17-16). *Perception forms* form the level of access between the management user and the software management data base; all queries of the data base are eventually effected using perception forms, so that the manager need never be aware of the actual data base structure. The programs and subroutines forming this level of access are intended to be standard across all developments, with interfaces defined and maintained by the secretariat function. Each project, however, has its own data base, and defines and maintains that data base.

User forms are mappable to and from perception forms, but not directly to and from the data base itself. One defining a user form need not have knowledge of the data base structure, only the structure of the perception forms. User forms are meant to be project-special, the software for processing such forms being provided by that individual project, or by the secretariat as a service.

Worksheets are the means by which the end user extracts information from other forms and operates on this information in a "scratchpad"

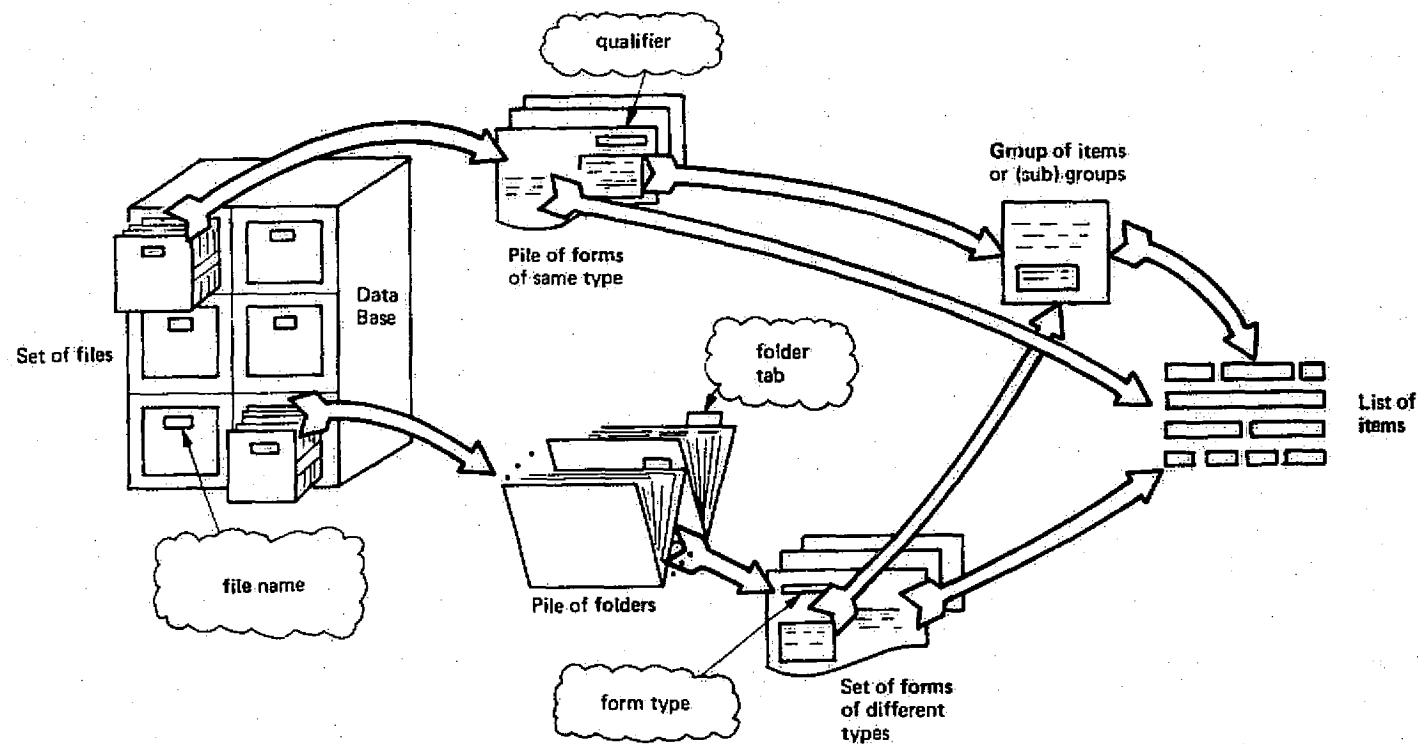


Figure 17-15. The end-user facility forms approach analogy

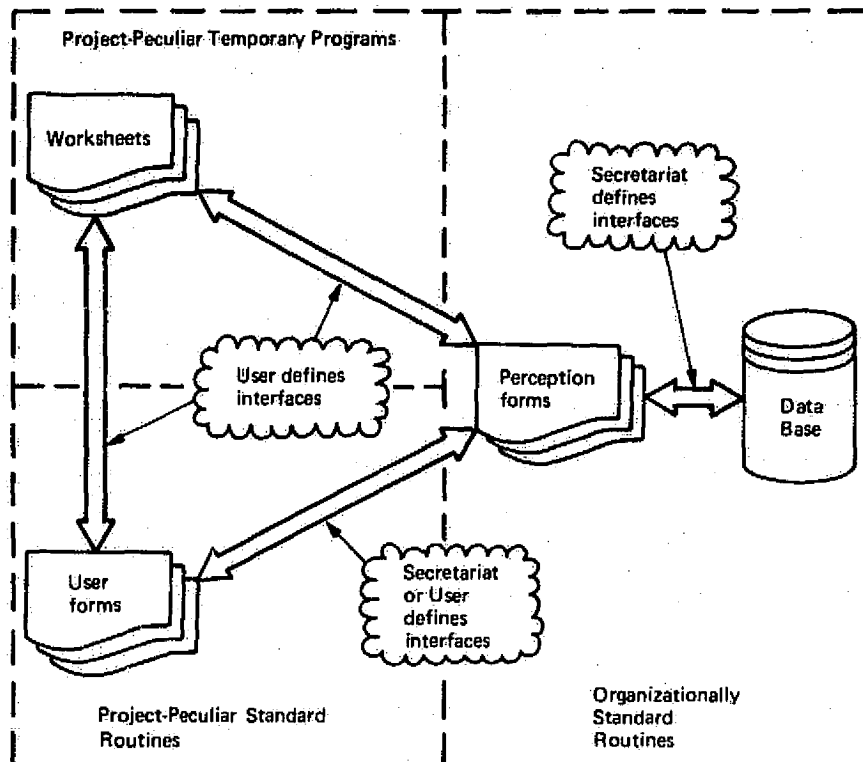


Figure 17-16. Interfaces in a forms-oriented end-user facility

fashion. Worksheets are programmed by each project or project manager as a temporary device for assembling and viewing data relative to team performance.

17.7.2 Management Information Processing

Generation of performance-monitoring reports extracts information from both the programming data base and the management data base proper. A management information system to accomplish the task needs to be flexible to meet the needs-range of its users and adaptable to reporting the status indicators of interest. Without value-to-measurement mappings, usage will probably be aimed primarily at spotting trends or making projections of costs, schedules, and the like.

Typical information (forms) of interest might include

- Tier chart and stub status, perhaps gauged relative to architectural baseline design
- Actual costs, computer usage, and manpower
- Programming transaction statistics
- Lines of code, data, documentation, etc.
- Schedule and PERT status and projections
- Decision Log
- List of action items, problems, anomalies, and dispositions
- Standards waiver log
- Work breakdown structure and performance status
- Configuration management data

The software to implement management forms constitutes what may be termed a "management information processor," and, inasmuch as it is manipulatable (programmable) by the user, it forms a "program management language." Users will be dealing with all five classes of the data object hierarchy shown in Figure 17-15: items, groups, forms, folders, and files. They will require naming and selection capabilities that direct the system to define and access occurrences of data in these classes. The class hierarchy makes this potentially very easy. For example,

TRANSACTION. SMITH. RUNTIME. MAY. 16

could refer to the TRANSACTION file, SMITH folder, RUNTIME form, MAY group, 16th (day) items.

Operations on forms to create worksheets may need to

- Define worksheet format and hierarchy
- Update management base
- Locate materials for query
- Sort/merge/extract material
- Move, copy materials within base
- Remove materials from base

The user of such a system must be willing to understand that his view of the data is largely predefined (by the secretariat) and that there is a finite set of such operations that he may invoke. Judiciously chosen, however,

these can make the performance evaluation task a much more informed process. The need for voluminous reports on a periodic basis may be drastically reduced, since managers can access the performance monitors in a timely manner and locate specific data of interest already in a report-like format.

17.8 CONCLUSION

I began this work with a promise to provide formal disciplines for increasing the probability of securing software that is characterizable by a high degree of initial correctness, readability, and maintainability, and to promote practices that aid in the consistent and orderly development of a total software system within budgetary and schedule constraints. Perhaps the initial chapters in this volume read as though the programmer himself were, thus, to become the programmed, a creative, but regimented, automaton.

Rather, the standards given represent a disciplined approach adaptable to the development of production software of almost any ilk. That discipline begins with the understanding why top-down, hierarchic, structured, modular methods offer the potential for programming improvement. It continues with the realization that the potential for improvement is no guarantee, but that justifiable adaptations or relaxation of standards on a case-by-case basis may be necessary and are not proscribed. It culminates in practice when the programming environment supports, and, indeed, fosters, the use of these standards.

The standard production system I have envisioned, more than described, in these final pages was meant to aid and abet the software engineer and his manager, and to profit his organization in consequence. Volumes could, and may yet, appear setting forth requirements, specifications, and usage of that system.

I would have liked to have addressed management data collection and reporting in more detail, as I would also with features for source data maintenance, data security, documentation support, language design, and many more topics so necessary in a production system well enough integrated and coordinated to be termed "Standard." However, the system philosophy, if not the detail, I think, comes through.

Before this monograph was begun, the development of quality software was truly an art, well-mastered by only a few of its many practitioners. The methodology reported in these pages has, more than once since, produced evidence that production programming may even now be a passing art

form. Its final passage will not be lamented, however, for it is being replaced by a more useful and effective engineering discipline. Whereas art forms are generally mastered by a few but, perhaps, appreciated by many, the engineering discipline will be both practiced and appreciated by an entire community of adherents. With that hope, I bring this work to a long overdue, but somewhat reluctant, close.

APPENDIX A

GLOSSARY OF TERMS AND ABBREVIATIONS

This appendix presents definitions of the major concepts and terms as used throughout this work. In general, concepts and terms found in an everyday non-technical vocabulary are not included; definitions given, which may also have a more general meaning outside the field of information processing, are herein oriented toward the more restricted context pertaining to software production and technology. Where possible, these definitions were made to conform to the ANSI vocabulary [42] and to usages in Webster's New Collegiate Dictionary [43] (oriented as above); inevitably, however, the definitions given may not be the same or equivalent to some current usages in the software industry.

Abnormal Termination. Termination of processing of a program or module within a program due to fatal errors which require that control be diverted to a program recovery mode, such as return to the user for subsequent decision making and manual operations.

Abstraction. A mechanism for hierarchic, stepwise refinement of detail by which it is possible at each stage of development to express only relevant details and to defer (and, indeed, hide) non-relevant details for later refinement.

Abstract Resource. Any commodity or available means that may be allocated toward the accomplishment of a task, characterizable by abstractions in representation, manipulations, and axiomatization.

Acceptance Criteria. Criteria that a set of software must satisfy in conformance with delivery requirements. Software delivered for interim operations with discrepant items is said to be accepted with "liens."

Acceptance Tests. Tests to verify acceptance criteria for program certification.

Accuracy. The degree of freedom from error; that is, degree of conformity to truth or to a rule. Contrast with precision.

Adaptation. Modification of existing software in order that it may be used as a module in a program development, as opposed to developing another module for that same purpose.

Algorithm. A prescribed set of well-defined rules or processes for the solution of a problem in a finite number of steps. In principle, the steps are sufficiently basic and definite that a human can compute according to the prescribed steps exactly and in a finite length of time, using pencil and paper. Contrast with heuristic.

Alphanumeric. Pertaining to a character set that contains letters and digits only. When applied to identifiers, it usually requires that the first character be a letter. A "spacer", such as the underscore character, is sometimes also admitted into the alphanumeric set for identifiers.

Arbiter. A mechanism for effecting the mutually exclusive use of a shared resource among concurrent processes.

Architectural Design. Selection among major alternatives relative to control logic and data structural topologies, module coupling modes, clocking, protocols, resource allocation strategies, etc., to that degree of detail which provides convincing evidence of production feasibility and which permits cost and schedule estimates of predefined accuracy.

Assessment of Correctness. The process of judging that a program (or part) is correct, based on a partial demonstration of its actual or envisioned behavior. Demonstration may range from rigorous, formal mathematical

proof to informal rationale, or from exhaustive testing to mere program checkout.

Audit. A formal or official examination that attests to the conformity (or non-conformity) between two supposed equivalent entities, according to a predefined set of rules.

Bottom-up Principle. A synthesis from concrete, low-level details by stepwise integration into higher-level capabilities or more abstract concepts as a method for solving a problem.

Calibration Error. An error purposely inserted into a program to serve as a means for gauging the completeness of testing to uncover indigenous errors.

Certification. A formal or official attestation that acceptance criteria have been met, or have been met subject to a noted set of liens for later removal.

Checkout. Informal validation of a program or a part of a program by members of the development team, once the item has been successfully compiled (or assembled), by running a series of tests. When such validation is performed only visually, the activity is referred to as "desk checking."

Coder. An individual mainly involved with writing but not designing a computer program.

Coding. The activity of expressing the steps of a given algorithm in a computer language (or, perhaps, more than one language). A unit is not qualified as "coded" until compiled (or assembled) and all syntax errors removed.

Cohesion or Module Strength. A relative measure of the strength of relationships among the internal components of a module insofar as they contribute to the variation in assumptions made by the outside program concerning the role the module plays in the program. Invariant assumptions about a module indicate high strength.

Competing Characteristics. A set of factors that relate to the final quality of a piece of software, but that may conflict or compete for project or machine resources. These may be ordered in priority to form implementation guidelines.

Complexity of a Program. The minimum (conceptual) length of the "proof of correctness" of a program, relative to a particular set of available methods for performing the "proof of correctness," such as formal mathematical rigorous theorem proving, informal (but complete) reasoning, exhaustive testing, etc.

Concurrent Processes. The simultaneous, overlapping, or time-interleaved operation execution of two or more processes in a single- or multiple-processing system.

Concurrent Production Principle. A method in which the formal production of software proceeds with concurrent activities among design, coding, testing, and documentation.

Confidence Level. The probability that a given statement concerning a set of random variables or a segment of a random process will be upheld, if tested.

Connections, Connectivity. The set of assumptions the rest of a program makes about a module (or other program segment). Modules have connections in control, in data, and in services (functions) performed. Connectivity increases with the number, type, and variability of such assumptions.

Consistency. A program quality which assures that the results of executing a program are repeatable in a practical sense, in spite of any logical errors which may be present in the program.

Control Data. Data that selects an operating mode or submode in a program, directs the sequential flow, or otherwise directly influences the function of a program.

Control Logic. The topological connectivity and the set of conditions that together govern the apparent sequencing of operations within a process (or among concurrent processes). Control logic is often displayed by means of a flowchart.

Correctness. Agreement between a program's total response and the stated response in the functional specification (functional correctness), and/or between the program as coded and the programming specification (algorithmic correctness).

Correctness Proof. Demonstration, however made, that the response of a program agrees with its functional and programming specifications.

CRISP, Control-Restrictive Instructions for Structured Programming. A set of keywords used to introduce structured control flow into an unstructured language. Also used as control sublanguage of CRISP-FLOW (flowcharts) and CRISP-PDL processors.

Critical Region. A region within a process in which a shared resource must only be accessed on a mutually exclusive basis for program consistency and correctness.

Data. Representations of measurements, observations, facts, statistics, or derived quantities, either actual, believed, or assumed, in a form suitable

for communication, reorganization, storage, retrieval, processing, and dissemination, Contrast with information.

Data Base. A collection of interrelated data items, usually stored together, to serve one or more applications by end users. As a goal, the data base has little (or controlled) redundancy, is stored so as to be independent of the usages made of its contents, yet is stored for optimal efficiency by such applications, and is capable of being subjected to a common and controlled approach when adding new data or when modifying or retrieving data existing within the data base. A "data base system" is a collection of separately structured data bases united by regulated interaction to form an organized whole.

Data Base Administrator. The custodian, an individual or organization, of a data base or data base system. The data base administrator does not own the data nor control its use; however, the data base administrator does control the structure and the modes of access of the data base(s), checks authority to use data in the base(s), and is responsible for the security and integrity of the data base(s).

Data Flow Diagram. A graphic representation that displays the paths of data through a problem solution. It defines the major or essential features of processing as well as the various media used.

Data Structure. A formalized representation of the ordering and accessibility relationships among stored data items, without regard to the actual storage configuration, as characterized by data-item types, ranges of values, and scope of activity, suitable for communication, interpretation, or processing by humans or automatic means.

Deadlock. That condition in which concurrent processes are each awaiting conditions that can never hold. Also called "stalemates" or "deadly embraces." Deadlocks occur when each of the deadlocked processes is waiting for the others to act, but each is unable to do so.

Debugging. Detection, location, and repair of inconsistencies between the program response and its functional or programming specification. A program or procedure is said to be debugged if no known anomalies are present.

Decision Table. A table of all or selected contingencies to be considered in the description of a problem or the specification of a solution, together with actions to be taken in each combination of contingencies. Also called "decision logic tables."

Design. That activity which defines program data structures and logical algorithms in response to, and conforming with, the software functional specification. It consists of describing the organization, data manipulations, I/O procedures and formats, etc., carried to a level of detail

sufficient for coding and operational implementation. Also, the word "design" may refer to the structure of the program resulting from the design activity, and, therefore, to the software programming specification. Development of the software functional specifications is sometimes called "functional design."

Deque. A data structure (Double-Ended *QUE*ue, pronounced "deck") that, together with its access functions, model operations with a linear list in which insertions and deletions can be made at either end of the list.

Development. That process by which new software comes into being as a process of design, rather than by a process of modification. It includes both the architectural and implementation phases.

Document. A medium and the information recorded on it for human use; by extension, any such record that has permanence and that can be read by man or machine. In this latter sense, a source code listing is a document.

Documentation. The creating, collecting, organizing, recording, storing, citing, and disseminating of documents or the descriptive material within documents.

Documentation Level. Specification of the degree of detail (A,B,C,D) and the format quality (1,2,3,4) for a particular item of documentation.

Engineering. Applied science concerned with the utilization of raw materials, products of technology, and physical laws for supplying human needs. A profession characterized by the propensity to solve technological and related problems with given constraints in an organized, responsible way.

Error. Any discrepancy between a computed, observed, or measured quantity and the true, specified, required, or theoretically correct value or condition.

Flag. A simple data structure that directs the flow of control in a program. If it has a range of only two values, it is sometimes called a "boolean" or "switch." Flags used solely to permit a program to have structured control flow are called structure flags.

Flowchart. A graphical representation for the definition, analysis, or solution of a problem in which symbols are used to represent operations. In this text, a flowchart describes the logic and sequence of operations in a module algorithm, drawn to conform to ANSI standards (Appendix B).

Function. Mathematically, a mapping between an input domain and an output range, in which each input has but a single image in the output. Therefore, in a program, a function performs a transformation or service as determined by its input. However, some modules that are not

functions in the mathematical sense (such as random-number generators and simulators of finite automata) are nevertheless described by "functional" specifications, because they perform well-defined actions as defined by finite-state machines. See *procedure*.

Functional Requirements Document (FRD). A document stating the essential technical features of a needed data processing capability, along with technical constraints and conditions to be met, and criteria for acceptable delivery that can be appended to or made a part of the SRD.

Functional Testing. Validation of program "functional correctness" by execution under controlled input stimuli. This testing also gauges the sensitivity of the program to variations of the input parameters.

Hardest-Out Principle. The building of a system beginning with that part which, in the final analysis, would have proved to possess the highest risk to programming if not performed first. At each subsequent step, the next *a posteriori* most critical part is added, until the entire software package is completed.

Heuristic. An exploratory method of problem solving in which solutions are discovered by evaluation of the progress made toward the final result. Contrast with *algorithm*.

Hierarchy. A structure by which objects or classes of objects are ranked according to some subordinating principle or set of principles. One common representation of a hierarchy is the directed tree-graph, in which the root node heads the hierarchy, and all other objects are ranked by order into levels of subordination. If a single subordinating relationship governs the hierarchy, it is said to be *unordered*; otherwise it is *ordered*.

HIPO. Hierarchic Input-Processing-Output descriptions. These descriptions, often viewed in the forms of graphics (so called HIPO charts), are used chiefly to express requirements and/or functional specifications for programs, routines, etc.

Identifier. A symbol whose purpose is to identify, indicate, name, or locate a data structure or procedure entry point.

Implementation. That process by which an architectural design is turned into a delivered program. It includes the detailed functional and procedural design, coding, testing, and documentation necessary to meet program requirements, either for new or modified software.

Indigenous Error. An error existing in a program (specification does not agree with performance) that has not been inserted for calibration purposes.

Inductive Assertion. An invariant predicate appearing within a procedure iteration. Usually placed just following the loop-collecting node, these predicates are used as an aid toward proving correctness.

Information. A representation of knowledge, intelligence, or other meaningful data in a form that can be used to cause or modify the purposeful actions of humans or machines, perhaps as the result of proper organization, analysis, and presentation. Contrast with data.

Information Structure. A representation of the elements of a problem or of an applicable solution method for a problem, insofar as its information base is concerned.

Information System. An assemblage of methods, techniques, procedures, programs, or devices that sense, convey, store, process, retrieve, or disseminate information, united by regulated interaction, to accomplish an organized, purposeful task.

Information Systems Technology. The body of knowledge and physical phenomena that constitute an applied science oriented toward the industrial usage of information systems.

Instructions. The repertoire of a language or (virtual) machine.

Integration. The combination of subunits into an overall unit or system by means of interfacing in order to provide an envisioned data processing capability.

Interface. When applied to a module, that set of assumptions made concerning the module by the remaining program or system in which it appears. Modules have control, data, and services interfaces.

Interface Testing. Validation that a module or set of modules operate within agreed interface specifications to assure proper data and logical communications.

Interrupt. Any stopping of a process in such a way that it can be resumed. A particular type of interrupt is the "trap."

Invocation. The linking to or insertion of a procedure body by means of a named reference within a procedure. Subroutine linking is sometimes referred to as a "call." Code insertion is referred to as a "macro call."

Level. The degree of subordination in a hierarchy. See also level of access and documentation level.

Level of Access. A set of functions, macros, subroutines, etc., that access a particular data structure or type of data structure, through which all accesses to that structure or type, except those within the functions, etc., must pass. Also called "clusters" and "Farnas modules."

Lexical Binding. Location of components constituting a module physically together.

Lien. A charge upon some discrepant software item in the form of a debt or duty later to be redeemed or otherwise satisfied. Usually this term refers to the delivery of software in some usable form but requiring the removal of discrepancies (program or documentation) in order to be complete.

Logic Error. An error in a program procedure, as opposed to an error in a program functional specification.

Look-Ahead Design Principle. The principle by which a baseline or preliminary design (or program architecture) is developed, which identifies and sketches the key details of the remaining work to be done to assure that the subsequent detailed implementation will be proper when viewed in retrospect.

Macro. A body of text substituted directly for a statement or portion of a statement recognized to be of a proper form. Macro invocations may transmit parameters for substitution or for processing before substitution into the code body that replaces the invocation.

Maintenance. Alterations to software during the post-delivery period in the form of sustaining engineering or modifications not requiring a reinitiation of the software development cycle.

Maximum Likelihood Estimator. That function of observed data that estimates an unknown parameter of a known or assumed probability distribution function as the value that maximizes the probability (density) function on the observed data.

Mode. A way of operating a program to perform a certain subset of the functions that the entire program can perform, as selected by control data or operating conditions. Often, the mode of a program will be defined as program states, with transitions annotated to delineate events causing the passages between modes of operation.

Modification. The process of altering a program and its specification so as to perform either a new task or a different but similar task. In all cases, the functional scope of a program under modification changes. Contrast with adaptation and sustaining engineering.

Module. Identifiable subportions of a program composed of instructions or statements in a form acceptable to a computer prepared to achieve a certain result. They are characterized by lexical binding, identifiable proper boundaries, named access, and named reference. The word "module" may apply to a subprogram, subroutine, routine, program, macro, or function. A "compile-module" is a module or set of modules

that are discrete and identifiable with respect to compiling, combining with other units, and loading.

Monitor. A level of access on a shared resource in a program with concurrent processes, including the means for arbitration of that resource.

Multiprocess. The simultaneous or concurrent execution of separate sequences of actions by multiple hardware processors.

Multiprogramming. The time-shared use of a processor in which two or more programs or program modules execute by interleaving operations in time.

Named Module. Modules which can be invoked by name (named access) and which internally may invoke submodules by name (named reference). Such invocation in the flowcharted design is denoted by the method of "striping" the flowchart symbol.

Nesting. The recursive application of the imbedding of structures (procedural or data) into a hierarchy of structural levels of definition.

Operating System. A system of routines and services that monitors, controls, allocates, de-allocates, and manages the execution of applications programs and other systems routines and their usages of system resources.

Operation. A well-defined finite-time execution within a program, performing a time-independent function based on its input.

Operator. (a) In a program, a function or characteristic action on data items (operands). (b) In computer utility, an individual who monitors or manipulates peripheral devices and input/output streams during the execution of a program or system.

Opossum. Any of a family of small American marsupials, chiefly nocturnal, largely arboreal, and almost omnivorous. When frightened, it feigns death [43].

Parallel Processes. The simultaneous or concurrent execution of two or more processes in devices such as multiple arithmetic units, logic units, or device channels.

Paranormal Termination. Unstructured escapes (in control) from a module in response to normal events or conditions. Compare with **abnormal termination**. Modules having paranormal terminations may yet exhibit a form of structured control flow, if properly configured into "paranormal extensions" of structured programming.

Perception Form. The level of access between end users and a computerized data base system in an end-user facility approach to data base operations. All user operations with the data base are eventually

effected using perception forms, as end users never need be aware of the actual storage structure of the data base. Such forms are usually standard across a number of end-user groups accessing the same or similar data bases. Perception forms are defined, programmed, and maintained by a data base administrator function.

Performance. A measure of the capacity of an individual or team to build software capabilities in specialized or generalized contexts. Performance distinguishes between work and effort, as it includes productivity as one component of its measure. However, performance also measures quality of work as measured by other criteria as well, as set forth in a prioritized list of "competing characteristics" early in development.

Phase of Production. That work related to the completion of a specified set of modules in conformance with requirements and goals. In top-down developments, a set of modules that are currently dummy stubs becomes the next implementation phase.

Post-order Traverse of a Graph (Tree). A method of "visiting" each node of a tree or other appropriate graph in which at each branching node the discipline, "visit each subgraph emanating from this node in leftmost order" and then "visit this node," is imposed recursively. The meaning of "visiting" is that particular action taken while at a given node.

Precision. A measure of the degree of discrimination with which a quantity can be stated, as opposed to accuracy, which states the degree to which that quantity is free from error.

Predicate. A logical proposition or assertion concerning the state of a program at a given point, having either a true or false value. Concerning program correctness, all such assertions must be axioms or proved true.

Pre-order Traverse of a Graph. A method of "visiting" each node of a tree or other appropriate graph in which at each branching node the discipline, "visit this node" and then "visit the subgraphs emanating from this node in leftmost order," is imposed recursively. The meaning of "visiting" is that particular action taken while at a given node.

Procedure. The course of action taken for the solution of a problem. A set of statements forming an algorithm. Procedures can be programmed as subprograms, subroutines, macros, or functions. A procedure may not, perhaps, always compute a function in the mathematical sense, but, nevertheless, the term "function" is often used to describe the characteristic action of such modules.

Procedure Design Language. A language for specifying algorithms in ordinary English or other language not to be compiled. Keywords usually appear, so as to format text and conform the specifications into a structured form. Also called a "Program Definition Language".

Process. A sequence of operations executed one at a time. Two processes are then concurrent if their operations can overlap or interleave arbitrarily in time.

Production. That portion of a software implementation that has to do with the generation of code and documentation and the checkout for correctness by production personnel. Production programming is characterized by the application of tradeoffs, known algorithms, and state-of-the-art solution methods toward software generation, as opposed to programming performed to extend the current state of the art.

Productivity. A measure of the rate at which individuals or teams can produce or have produced software items, or can perform or have performed software-related tasks.

Program. A series of instructions or statements, in a form acceptable to a computer, prepared to achieve a certain result, to perform a specified function within a subsystem.

Programming. A generic word sometimes used to describe the overall process of program design, coding, and testing, but often it is used to connote only coding and checkout.

Programming Specification (PS). That portion of the Software Specification Document (SSD) which sets forth descriptions of algorithms, data structures, the modular definition, etc., in sufficient detail that the program can be coded without functional or algorithmic ambiguity.

Proper Program. A program or program segment, such as a subroutine, subprogram, or function, which has but one point of entry (in control) and but one mode of exit (although, if a subroutine, it may be called from, and return to, many points in a program).

Protocol. A rule prescribing the interface disciplines and correct procedures for communications with a program, subroutine, operating system, or hardware device.

Queue. A data structure which, together with its access functions, models operations used in first-in first-out (FIFO) list algorithms.

Real-Time Process. A process actuated by and acting in response to an external event sensed by the computer.

Reliability Index. The probability that a program or device will perform without failure for a specified period of time or amount of usage.

Requirement. A characterization of the essential features of a needed data-processing capability, along with a set of constraints and conditions to be met.

Requirements Testing. Execution of a program under controlled conditions to demonstrate that all stated or implied requirements and performance criteria have been met.

Resource. Any commodity or available means that may be allocated toward the accomplishment of a task. In concurrent processes, shared resources are characterized as "devoted" (allocated for mutually exclusive use) or "mutual" (can be engaged in simultaneous operations under stated limitations). Resources produced by one process and consumed by another are said to be "temporary resources"; other resources are "permanent."

Routine. A program or program module that may have some general or frequent use. If a program module, then a routine always returns to the point of invocation after execution (i.e., a proper routine) or abnormally terminates. Other types of routines are not discussed in this work.

Scope. The range within which an identified unit displays itself. Scope of activity refers to the boundaries within which a data structure or program element remains an integral unit. Scope of control refers to the submodules in a program that potentially may execute if control is given to a cited module. Scope of error denotes the set of submodules that are potentially affected by the detection of a fault in a cited module.

Secretariat. A centralized facility consisting of processing aids, library materials, and production services available to development projects, for the purpose of raising productivity, enforcing standards, and monitoring progress.

Semantics. The set of rules that defines relationships between symbols and their meanings; in computer language, the rules that define what effects are caused by statements in the language.

Semaphore. A shared data structure used by concurrent processes to effect synchronization, consisting of an arbitrated variable that contains the net number of "messages" sent, not yet received, and a queue that contains a list (if not empty) of processes currently waiting for a "message."

Sequential Testing Procedure (STP). A procedure for searching a decision table condition entry to determine which rule applies to a given array of answers to the condition stub. Thus, an algorithm for processing the upper half of a decision table to determine which rule is to be activated.

Side-Effect. A secondary effect due to connectivity among modules which, therefore, propagates in the same mode as program connections: control, data, services. Control side effects arise in non-proper programming; data connection side effects arise in use of COMMON, external coupling, content coupling, and not utilizing the normal parameter passing mechanism; and service side effects arise when the role or action of a

procedure varies with its application (as a result of global variables modified, local data modified and retained, or changes in hardware status).

Software. A computer program (or set of programs), together with all materials, procedures, and documentation concerned with the use, operation, and maintenance of a data processing capability.

Software Design Definition (SDD). A document chiefly used to display the results of the architectural design study and implications of cost, schedule, and work-breakdown structures to management or customers. Some high-level technical material, such as that needed to assess problem areas and other concerns or to show how requirements will be met, is included.

Software Development Library (SDL). A project internal facility for interface and management visibility and for software production management and control.

Software Engineering Management. The judicious use of means to effect and administer the advancement or usage of information systems technology. Software engineering management recognizes needs, sets goals, plans modes of accomplishment, devises means for resource allocation, and directs the approach taken in future information systems applications and in the solution of problems associated with these applications.

Software Functional Specification (SFS). That part of the Software Specification Document (SSD) which defines the end-to-end functional response of the program in terms of input stimuli, program behavior, and output contents. Any program that conforms to the SFS is deemed functionally correct.

Software Requirements Document (SRD). A document chiefly generated by a customer or other initiator used to display the needs, justification, and estimated costs associated with the implementation of a data-processing capability. Some technical material, such as that needed in support of the justification or establishment of needs, is included. Detailed technical requirements may be appended or included in the Functional Requirements Document (FRD) portion of the SRD.

Software Specification Document (SSD). The principal program documentation produced by a development project, consisting of "as-built" Functional (SFS), Programming (PS), and Test Specifications.

Software Test Report (STR). A report of the tests and the results of tests performed in demonstration of delivery requirements.

Specification. A statement or set of statements (documentation) containing a detailed description or enumeration of particulars with respect to the function or construction of a piece of software.

Stack. A data structure that, together with its access functions, models operations used in last-in first-out (LIFO) list algorithms.

Standard. That which is set up and established by authority, custom, or general consent as a model, example, criterion, test, or rule for the definition or measure of quantity, weight, extent, value, or quality.

Strength. See cohesion.

Striped Module. A named module in the program procedural design, so called because of the method used to denote such modules on a flowchart. Striping of a flowchart symbol signifies that a detailed representation is either located elsewhere in the same set of flowcharts (horizontal striping), or else at a referenced location (vertical striping).

Structure. May pertain to the manner or form in which something is constructed or may refer to the actual system as constructed. Descriptions of structure focus an interrelation of the various parts as dominated by the general character or function of the whole. Designing structure is a process of identifying, analyzing, and selecting among alternatives within design categories.

Structured Program. As used in this work, a program whose control logic topology adheres to strict rules of form, being composed of iterations and nestings of a set of basic planar flowchart constructions IFTHENELSE, WHILEDO, DOWHILE, CASE, and sequence formats, along with, perhaps, paranormal extensions.

Structure Flag. A flag introduced into an otherwise unstructured program to permit structured control flow.

Structure Graph. A graphical representation showing the control connections between named modules. The "top" node of the graph represents the top-level main program procedure; lines from the top node to other nodes signify that the corresponding named modules appear as invocations in the top-level program procedure, etc.

Stub, Dummy. Segments of temporary code that replace named modules for purposes of correctness testing a program during top-down construction. These are usually simple procedures that merely test interfaces or supply test inputs for an algorithm under test. The actual procedures replace the dummy stubs as they are constructed.

Suave. (adj) Blandly pleasing; smoothly polite; urbane; polished. Synonyms: suave, urbane, diplomatic, bland, smooth, politic mean ingratiatingly

tactful and well-mannered. Suave specifically suggests the power to encourage easy and frictionless intercourse [43].

Submodule. A module appearing within a module or invoked by a module. On a flowchart, the procedure appearing within or referred to (e.g., invoked by) any charted symbol.

Subprogram. As used here, a module whose invocation appears but once in a procedural specification of a program. Subprograms may be coded inline, as a macro (used once), or so as to be linked to (and from) as a separate procedure body.

Subroutine. A routine that can be a part of a routine. As used here, a subroutine is always a proper procedure (one entry point, with return only to point of invocation), and appears in more than one invocation in the program procedure. Some subroutines may also have abnormal terminations.

Sustaining Engineering. Software-related activities in the post-delivery period, principally supportive in form, which keep that software operational within its functional specifications; e.g., repairing faults, correcting documentation, removing liens, and estimating costs and other resources required for such tasks. The holding or keeping of software in a state of efficiency or validity despite interface fluctuations in system, subsystem, or applications capabilities.

Synchronization. The scheme by which arbitration constrains the ordering of operations on shared resources among concurrent processes in time so as to enable consistency in the program behavior.

Syntax. The set of rules that defines the valid input strings (sentential forms) of a computer language as accepted by its compiler (or assembler). Therefore, the structure of expressions in a language, or the rules governing the structure of a language.

System. An assembly of methods, procedures, programs, or techniques united by regulated interaction to form an organized whole.

Testing. Execution of a program (or partial program), usually under a controlled set of input conditions, program configurations, and input stimuli, performed in order to observe the actual program response.

Tier Chart. A tree-graph representation of a program and its named modules, in which the subordination relation is invocation. Subroutine invocation nodes occur more than once; however, all but one of these nodes appear as leaves of the tree, and the other forms the root of the subroutine tier hierarchy.

Top-Down Principle. A synthesis from abstract, high-level concepts by stepwise refinement into lower-level concepts and details, as a method for solving a problem.

Topological Sorting. An algorithm for listing the nodes of a directed acyclic graph, in which precedence in the graph implies precedence in the list. One such algorithm is the following: Locate a node having no edges directed into it, transfer it to the list, and delete all edges emanating from it; repeat until all nodes have been listed.

Trap. A special form of program interrupt: an unprogrammed conditional jump to a known location, automatically actuated by hardware, with the location from which the jump occurred recorded.

Tree. An acyclic connected graph. If the tree has $n \geq 2$ nodes, then it also has $n - 1$ edges. Every pair of nodes is connected by exactly one path. As used in this work, the tree often represents a hierarchy, in which edges are directed to denote a subordinating relationship between the two joined nodes.

Type (Data). A set of attributes used to define a set whose elements are data structures and on which an algebra is defined. Fundamental types are those explicit in a programming language. Fundamental simple types usually include integers and reals, and fundamental structures usually include the indexed array.

User. An individual or organization which normally supplies information for processing, or which normally receives, interprets, and utilizes the output of such processing. Contrast with operator.

User Forms. Forms mappable to and from perception forms in an end-user facility approach to data base operations. User forms are standard within an end-user group accessing a data base that has been standardized over even a wider set of user groups. User forms are defined by the end-user group, and may be programmed and maintained by either the end-user group or the data base administrator.

Validation. Demonstration that a software item or implementation conforms to its specification (whether the specification is correct or not). Validation attests primarily to conformity with the grounds on which something is based, to a greater extent than conformity with accuracy criteria. Compare with verification.

Verification. Confirmation that a program has accurately satisfied acceptance criteria, or that a data transcription or other such operation has been accomplished accurately. Verification primarily deals with accuracy (freedom from error) to a greater extent than conformance to a design. See validation.

Work Breakdown Structure. An enumeration of all work activities in hierarchic refinements of detail that defines work to be done into short, manageable tasks with quantifiable inputs, outputs, schedules, and assigned responsibilities. It is used for project budgeting of time and resources down to the individual task level, and as a basis for progress reporting relative to meaningful management milestones.

Worksheet Forms. Forms defined, programmed, and maintained by the end user in an end-user facility approach to data base operations. Worksheet forms are mappable to and from user forms and perception forms. They are used for extracting, assembling, and viewing of data by the end user in a "scratch pad" fashion, as a (perhaps) temporary device pertinent to that user.

APPENDIX B

STANDARD FLOWCHART SYMBOLS

Symbols are used on a flowchart to represent the functions of an information processing system. These functions are input/output, processing, flow path and direction, and annotation.

A basic symbol is established for each function and can always be used to represent that function. Specialized symbols are established that may be used in place of a basic symbol to give additional information.

The symbols given in this appendix derive from *American National Standard Flowchart Symbols and Their Usage in Information Processing*, ANSI-X3.5-1970, American National Standards Institute, Inc., New York, Sept. 1, 1970. The ANSI usage, in some cases, has been refined or extended herein to conform to the needs of structured programming.

Table B-1. Basic flowchart symbols





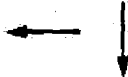
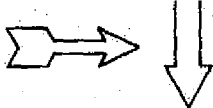
Symbol and dimensional ratio (width:height)	Meaning
 1:2/3	Input/Output Symbol. This symbol represents an I/O medium or function, such as making available of information for processing (input), or the recording of processed information (output).
 1:2/3	Process Symbol. This symbol represents any kind of processing; for example, the process of executing a defined operation or group of operations resulting in a change in value, form, or location of information.
 1:2/3	Decision Symbol. This symbol represents a specific decision or switch operation that determines which of a number of alternate paths is to be followed. This symbol may not be striped.
 1:2/3	Comment or Annotation. This symbol is used to enclose descriptive comments or explanatory notes as clarification. The broken line is connected to any symbol where the annotation is meaningful.
	Sequence, Control Flow. Sequential program flow is indicated by single-line arrows connecting symbols. Arrowheads are necessary to show direction. Use only one arrowhead per flowline, at its end.
	Information, Data Flow. Flow of information or data is indicated by double-line arrows connecting symbols. Arrowheads are necessary to show direction. Tails are optional, but recommended.

Table B-1. Basic flowchart symbols (continuation)




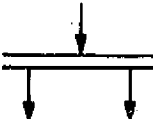
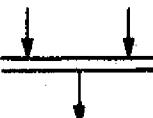

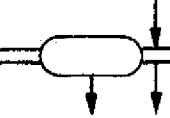
Symbol and dimensional ratio (width:height)	Meaning
 1:1	Loop Collecting Node. The small open circle shown represents the iteration point in a looping operation.
  1:1	Decision Collecting Node. The small blackened circle shown represents the merging of alternative flow paths in a program. If this symbol is computer drawn, it may be desirable not to have the circle completely blackened, but filled, say, with an asterisk, as shown.
	Begin Concurrent Mode, or Fork. The symbol shown represents the beginning of two or more concurrent (parallel or interleaved) processes.
	End Concurrent Mode, or Join. The symbol shown marks the end of two or more concurrent (parallel or interleaved) processes. When used to join background and interrupt processes, the interrupt logic is disabled on resumption of sequential mode.
 1:3/8	Terminal Symbol. The symbol shown represents the entry or exit point of a flowchart.
 1:3/8	Interrupt Symbol. The symbol shown represents the enabling (or arming) of an interrupt that may initiate a concurrent (preemptive interleaved) process. The process reexecutes each time the event occurs until both processes reach their join. The event identifier (name) is placed in the terminal symbol.

Table B-1. Basic flowchart symbols (continuation)

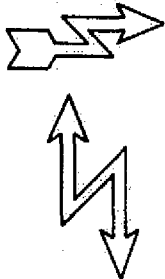
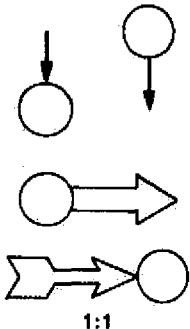
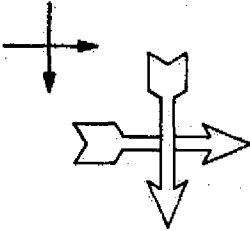
Symbol and dimensional ratio (width:height)	Meaning
	<p>Communication Link. The symbol shown represents a function in which information is transmitted by a telecommunication link. Arrowheads are necessary to show direction. Tails are optional, but recommended.</p>
	<p>Connectors. The symbols shown represent out-connectors and in-connectors for control and data flow. A set of two such connectors represents a continued flow direction when the flow is broken by any limitation of the flowchart. The use of such connectors to off-page continuations is discouraged except for CASE structures with too many branches to fit on one page.</p>
	<p>Crossing of Flow Paths. Flowlines may cross; this means they have no logical interrelation. Crossing of control flowlines is discouraged except when absolutely necessary.</p>

Table B-2. Specialized I/O flowchart symbols

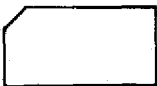



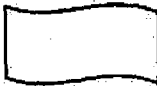
Symbol and dimensional ratio (width:height)	Meaning
 1:1/2	Punched Card. The symbol shown represents an I/O operation in which the medium is punched cards.
 5/4:2/3	Deck or File of Cards.
 1:2/3	Online Storage. The symbol shown represents an I/O function utilizing any type of online storage, such as magnetic tape, drum, or disk.
 1:1	Magnetic Tape. The symbol shown represents an I/O function in which the medium is magnetic tape.
 1:1/2	Punched Tape. The symbol shown represents an I/O function in which the medium is paper tape.

Table B-2. Specialized I/O flowchart symbols (continuation)

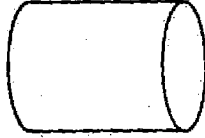
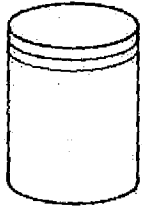
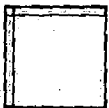

Symbol and dimensional ratio (width:height)	Meaning
 5/4:2/3	Magnetic Drum. The symbol shown represents an I/O function in which the medium is a magnetic drum.
 2/3:5/4	Magnetic Disk. The symbol shown represents an I/O function in which the medium is a magnetic disk.
 1:1	Core. The symbol shown represents an I/O function in which the medium is core storage.
 1:2/3	Document. The symbol shown represents an I/O function in which the medium is a document. It is used often to denote output of hard-copy material on either line printers or typewriter terminals.

Table B-2. Specialized I/O flowchart symbols (continuation)



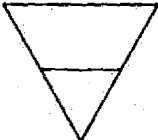
Symbol and dimensional ratio (width:height)	Meaning
 1:1/2	Manual Input. The symbol shown represents an input function in which information is entered manually during processing, such as by online keyboards, switches, or push-buttons.
 1:2/3	Display. The symbol shown represents an I/O function in which the information is displayed for human use at the time of processing, by means of online indicators, video devices, console printer, plotters, etc.
 1:0.866	Offline Storage. The symbol shown represents the function of storing information offline, regardless of the medium on which the information is recorded.

Table B-3. Specialized process symbols




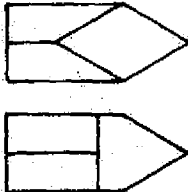

Symbol and dimensional ratio (width:height)	Meaning
 1:2/3	Internal Procedure. The symbol shown represents a named procedure (subprogram or subroutine) module that has a more detailed representation elsewhere in the same set of flowcharts. Similar horizontal striping conventions apply to other symbols, as well, when they are detailed in this way.
 1:2/3	External Procedure. The symbol shown represents a named procedure (subroutine) module or logical unit that is not detailed in this same set of flowcharts. Similar vertical striping conventions apply to other symbols, as well, when they are detailed elsewhere.
 1:2/3	Preparation. The symbol shown represents the preparation of a medium for processing, such as obtaining core storage, declaring data structures, or initializing variables.
 1:4/9	Indexed Looping. The symbols shown represent loop initialization, predicate testing, and update functions. Testing always follows every initialization and update.
 1:4/10	Non-normal Exit. The symbol shown represents the exit from a process due to abnormal or paranormal events.

Table B-3. Specialized process symbols (continuation)







Symbol and dimensional ratio (width:height)	Meaning
 1:0.866	Merge. The symbol shown represent the combining of two or more sets of items into one set.
 1:0.866	Extract. The symbol shown represents the removal of one or more specific sets of items from a single set of items.
 1:1.732	Sort. The symbol shown represents the arranging of a set of items into a particular sequence.
 1:1.732	Collate. The symbol shown represents merging with extracting; that is, the formation of two or more sets of items from two or more other sets.
 1:2/3	Manual Operation. The symbol shown represents any offline process geared to the speed of the human being without using mechanical aid.
 1:1	Auxiliary Operation. The symbol shown represents an offline operation on equipment not under direct control of the central processor.

Table B-4. Symbol usage in flowcharting


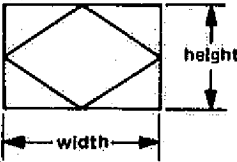
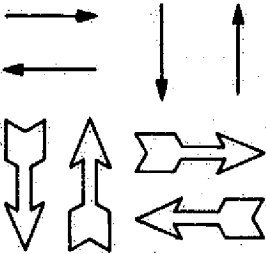
Illustration	Usage
	<p>Symbol Shape. The actual shapes of the symbols used should conform closely enough to those shown in Tables B-1, B-2, and B-3 to preserve the characteristics of the symbol. The curvature of the lines and the angles formed by the lines may vary slightly from those shown in this standard, as long as the shapes retain their uniqueness.</p>
	<p>Symbol Size. Flowchart symbols are distinguished on the basis of shape, proportion, and size in relation to other symbols. Proportion of a given symbol is defined by the rectangle in which that symbol can be inscribed. Dimension and relative size of the rectangles are given with each symbol by a pair of numbers (width: height).</p> <p>The size of each symbol may vary, but the dimensional ratio of each symbol shall be maintained.</p>
	<p>Symbol Orientation. The orientation of each symbol on a flowchart should be the same as shown in the tables of this appendix. Flowline symbols (either control, information, or data flow) may be drawn left-to-right, top-to-bottom, right-to-left, or bottom-to-top. The principal flow of control is top-to-bottom. The principal flow of information or data should be depicted as left-to-right.</p>

Table B-4. Symbol usage in flowcharting (continuation)

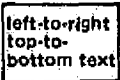
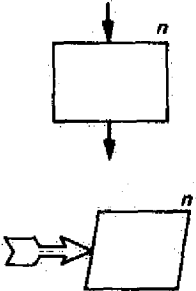
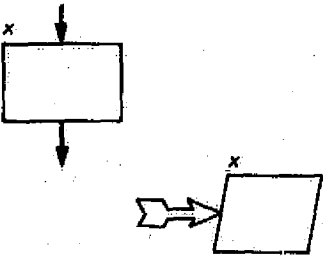
Illustration	Usage
	<p>Flowchart Text. Descriptive information within each symbol shall be presented so as to be readable from left-to-right, top-to-bottom, regardless of the direction of flow outside the box.</p>
	<p>Symbol Identification. The identifying number n assigned to a symbol on the current flow chart shall be placed above and to the right of its vertical bisector. This number, concatenated with the chart identifier c, forms the unique symbol Dewey-decimal identifier, $c.n$. If the symbol is striped and if there is no explicit symbol cross-reference, then the number $c.n$ becomes the (implicit) cross-referencing chart number at the next hierarchic level.</p>
	<p>Symbol Cross-Reference. The identifying chart number or other cross-referencing element x shall be placed above the symbol and to the left of its vertical bisector. When such notation appears, it takes precedence over the symbol identifier as the cross-referencing scheme.</p>

Table B-4. Symbol usage in flowcharting (continuation)

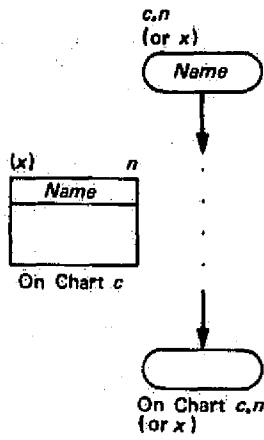
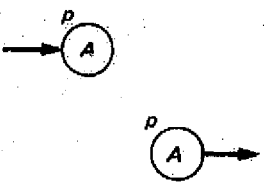
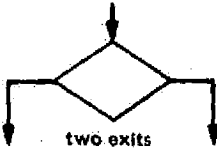
Illustration	Usage
	<p>Symbol Stripping. A horizontal line is drawn within, completely across, and near the top of the symbol, and a reference to the detailed representation is placed between that line and the top of the symbol. The terminal symbol shall be used as the first and last symbols of the detailed representation. The entry terminal symbol contains the name reference that also appears in the striped symbol.</p> <p>The location of the detailed representation chart is contained in the symbol cross-reference (x), if any; otherwise it is referenced by the concatenation of the chart and symbol identifiers.</p> <p>In either case, the chart number is placed above and to the left of the vertical bisector of the entry terminal symbol.</p>
	<p>Connector Identification. A common identifier, such as an alphabetic character, number, or mnemonic label (A) is placed within the connector as shown. Additional cross-referencing for off-page connectors shall be the page number, p, placed above and to the left of the vertical bisector of the symbol (only when there is more than one such page). Off-page connectors to or from flowcharts of other modules are not permitted.</p>

Table B-4. Symbol usage in flowcharting (continuation)

Illustration	Usage
 <p>two exits</p>	<p>Multiple Control Flow Branches. Multiple branches from a symbol are restricted to the decision symbol. The text within the symbol shall state the explicit predicate or event that causes the branch.</p> <p>For each conditioned branch, each exiting flowline is to be labeled by text that identifies the predicate outcome. Normally, <i>true</i> exits to the left, <i>false</i> to the right in binary decisions; multiple branches exit in case-order from the left (if there is an explicit case order).</p> <p>Event-actuated branches need only annotate exiting flowlines when there are multiple outcomes to a stated event within the decision symbol.</p>

REPLACING PAGE BLANK NOT FILLING

APPENDIX C

SOFTWARE REQUIREMENTS DOCUMENT TOPICS

This appendix contains a detailed outline for the organization of software requirements suitable for hierarchic refinement of detail. The first sections contain preliminary management information and the others go into more technical detail, as needed.

Figure C-1 is a top-level, visual table of contents of the document organization. The suggested topics are then detailed in the remainder of this appendix. Each of the topical headings is followed by a narrative description for the material to be inserted.

Much of this material may, perhaps, be supplied by reference or by attachment. Such practice is to be encouraged, inasmuch as the SRD is not generally a document that is maintained after program delivery.

The main principle behind the writing of the SRD is that it gives management only what it needs (the SJR) in order to approve the expenditure of funds and to concur with the schedule—at least for the initial (SDD) study phase of the implementation—and gives implementation only what it needs (the FRD) in order to respond with an appropriate SDD.

See Chapter 11 for more specific and detailed rules in completing the SRD.

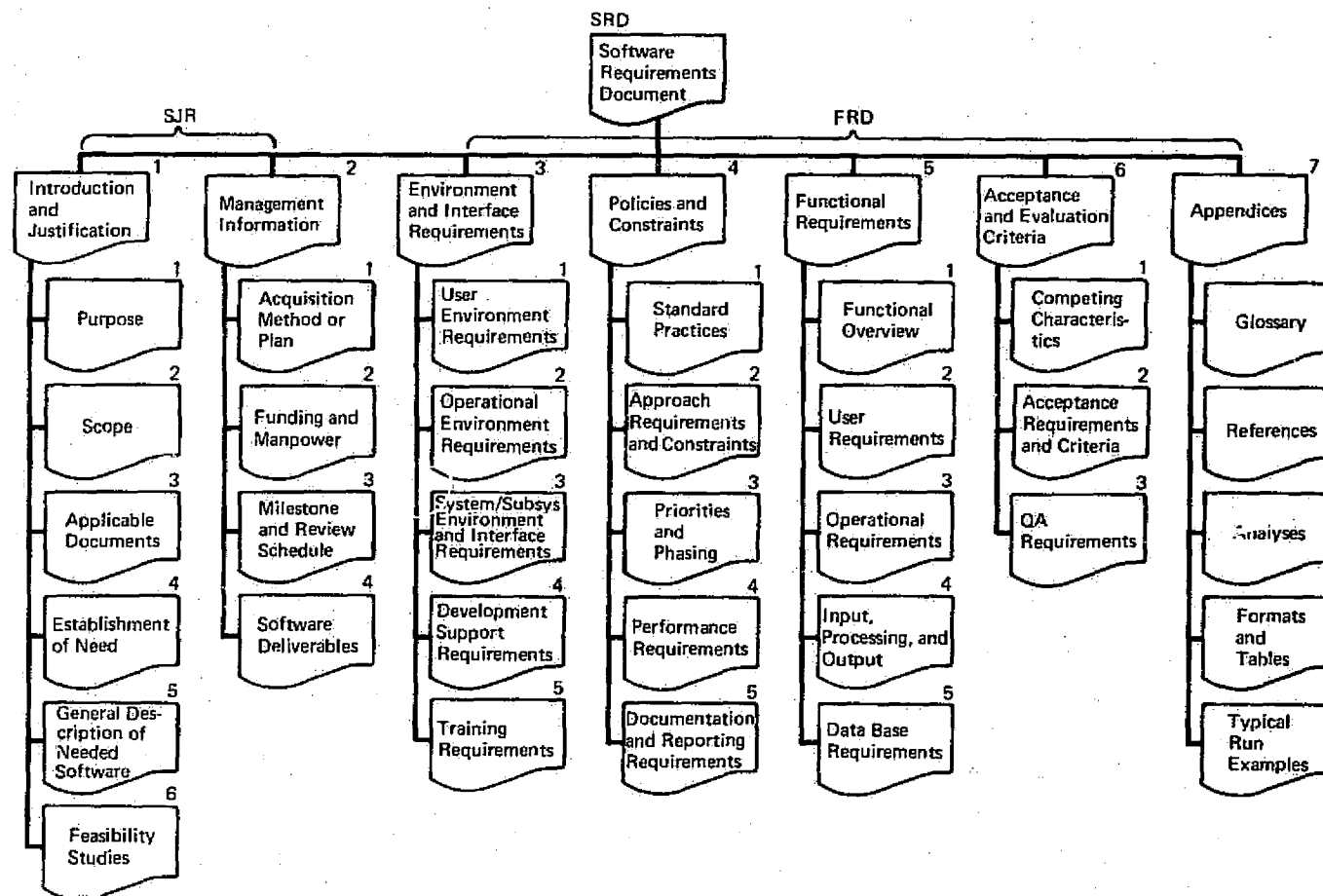


Figure C-1. A hierarchic outline for the SRD

SOFTWARE REQUIREMENTS DOCUMENT

Detailed Table of Contents

FRONT MATTER

Title Page. Provide a title page containing (1) document number; (2) **SOFTWARE REQUIREMENTS DOCUMENT**; (3) program, project, subsystem, and system titles; (4) release date; (5) a signature-approval block appropriate for the management authority level required; and (6) releasing organization. Signatures, when supplied, signal the approval to proceed with the architectural phase. FRD material may precede SRD approval, or may be supplied subsequent to approval.

Abstract. Give a brief abstract that summarizes the program requirements, including costs, resources, and schedule.

Change Control Information. Provide a brief statement that identifies the method for approval and for arbitration of changes in the SRD.

Distribution List. Provide a distribution list of all parties who are to receive copies of the SRD.

Distribution Information. Provide a statement that tells how additional copies of the SRD may be obtained.

Table of Contents. Provide a detailed table of contents for the SRD, which lists section number, topic title, and page of every item with a heading. (This is probably the last part of SRD to be completed.)

TEXT OF SOFTWARE REQUIREMENTS DOCUMENT

1. INTRODUCTION AND JUSTIFICATION

1.1 Purpose of This Requirement

Provide a brief general description of the software to be provided, its purpose, and the user benefits or mission characteristics upon which the requirements for this software are established.

1.2 Scope of Applicability

Summarize the level and type of material that appears in the SRD. Identify the intended readers. State the criteria that govern the content, extent, and format of this SRD. If pertinent, proscribe items specifically to be outside the scope of this application. Reference applicable or governing documents as appropriate.

1.3 Applicable Documents

Identify all documents that establish technical requirements, set constraints and policies, or regulate the procedures to be used in implementation.

1.4 Establishment of Need

1.4.1 History and Requirement

State specific user needs, mission requirements, market potential, or other goals to be satisfied through the services provided by the software. If pertinent, trace the historical development of the application to be automated.

1.4.2 Analysis of Current Capability

Describe any current capability or in-progress developments that respond wholly, or in part, to the needs stated in 1.4.1. Identify needs for which no current capability exists, and give limitations on capabilities that do exist. State costs, manpower, and other performance figures for providing or maintaining current services, if these are pertinent to the establishment of the need.

1.4.3 Analysis of Required Capability

Present justification for the development of software to fulfill the requirements above, based on estimated performance gains as they relate to accomplishing the stated goals of the approving organization.

1.5 General Description of the Needed Software

Provide an introductory description of the software to be developed, and show how this software will meet the needs in 1.4. Use descriptions as appropriate to characterize the program, to provide tradeoff studies, to show interrelations between the program and its application, and to display its major operational features.

Identify the expected program life, whether the program will be a developmental, interim, or operational fulfillment of the need, and estimate major characteristics, such as execution mode (real-time, interactive, batch),

computational complexity (number-crunching or data manipulative), etc. Use graphic aids to illustrate these features.

1.6 Feasibility Studies

Evaluate the proposed software, considering timeliness, technological and economic feasibility, and capability. Identify major risks that will require later resolution.

2. MANAGEMENT INFORMATION

2.1 Acquisition Method or Plan

2.1.1 Method of Acquisition

State the method to be used to develop the software, such as in-house, outside contract or some mixture of the two, e.g., in-house definition and design with outside coding and checkout. If done by outside contract, state the procurement cycle lead times and other constraints pertinent to outside contract or purchase of software relative to this development activity.

2.1.2 Responsible Group and Staff

Identify the in-house organization responsible for the software development and identify cognizant individuals as appropriate.

2.2 Funding and Manpower

2.2.1 Total Project Estimates

State the amount and source of funding to be allocated to manpower, procurements, and services, for both in-house and outside-contract parts of the development. State methods used to estimate these, and state the probable accuracy in the estimates.

2.2.2 Architectural Phase Estimate

Detail the manpower and funding required for the architectural study phase of development. Place limits on the expenditures during this phase that are not to be exceeded.

2.3 Milestone and Review Schedule

Provide a milestone schedule showing estimated major tasks, priority phasing, review milestones, and software deliveries. State methods used to estimate schedule events. If PERT is used, state critical-path parameters. Set milestone and probable accuracy by requirements keyed to mission objectives. Show the architectural phase in detail, and state not-to-be-exceeded limits on the completion of this phase.

2.4 Software Deliverables

Provide a checklist of specific items required for delivery. If delivery is phased or prioritized, then state any requirements that define deliverable items relative to phases or priorities.

3. ENVIRONMENT AND INTERFACE REQUIREMENTS

Introduce the general environment within which the program is to operate, in which are identified the users, operators, and maintenance personnel, the envisioned or required system, and the interfaces with each of these. If pertinent, describe the working environment of personnel.

Identify whether the required environment is currently existing, or whether portions will have to be purchased, leased, or otherwise created, either on a dedicated or time-shared basis.

Illustrate the environmental requirements graphically as well as verbally. Show the flow of data or paperwork from the users through the application.

Use hierarchic levels of detail provided below as appropriate to characterize the environment (required or existing), to provide tradeoff analyses, to show interrelations between environmental attributes, and to display the major environmental features.

3.1 User Environment Requirements

Identify and describe assumed or required user interfaces, such as who the users are, where they are located, what their sources of data are, how they will generate and submit data to the program or request runs, how they will make use of the output, other manual tasks, etc. Defer user procedures, formats, units, etc., until Section 5.2 (User Requirements).

3.2 Operational Environment Requirements

Identify and describe assumed or required operator interfaces, such as control devices, input devices, operator procedures, data generation methods, means for delivery of output to users, manual tasks, etc. Identify those requirements that are unique to this program and not covered by an overall system requirement or governing document. Defer operational procedures, formats, units, etc., until Section 5.3 (Operational Requirements). State any operational requirements for security, privacy, and protection of program resources.

3.3 System/Subsystem Environment and Interface Requirements

Provide a narrative overview of the total system in which the program is required or will be assumed to operate. Include graphic material as

appropriate, such as a block diagram of the pertinent system hardware and software.

3.3.1 Hardware Characteristics and Constraints

Describe the hardware resources that may be assumed or are required as the starting point for program development.

3.3.2 Software Characteristics and Constraints

Describe the software environment in which the program is to operate, and identify required software resources and other constraints on the development, such as required interfaces with existing user programs, data bases, compilers, diagnostics, system software, etc. Identify known documents and manuals required for program development. Defer details of units, formats, media, etc., until Section 5.4.

3.4 Development Support Interfaces and Requirements

Identify and describe any special interfaces required during development, as well as requirements for any supporting facilities or resources, such as staffing, services, special software or hardware, computer time, manuals or documents, logistics, etc., which are applicable to the development activity but not covered by 3.1, 3.2, or 3.3.

3.5 Training Requirements

Describe any training requirements that may impact program development, conditions for delivery, or later program maintenance and operation.

4. POLICIES AND CONSTRAINTS

4.1 Standard Practices, Policies, and Procedures

4.1.1 Established Development Practices, Policies, and Procedures

Identify any existing appropriate practices, policies, and procedures or controlling documents that will be applicable to, and required of, the development of the program.

4.1.2 Exceptions to Established Practices, Policies, and Procedures

Identify extensions, modifications, additions, or other exceptions to established policies in 4.1.1, above. Include a description of any special practices, policies, or procedures applicable to, and required of, the development of the program.

4.2 Approach Requirements and Constraints

4.2.1 Development Philosophy

Include a brief description of the philosophy, principles, or disciplines required in developing the program.

4.2.2 Development Constraints

Identify any constraints to be placed on the development approach, such as design medium, programming language, maximum core size, required speed of execution, use of available subroutines, special interface design procedures, etc.

4.3 Priorities and Phasing

Define priorities of requirements and give guidelines governing how resources may be allocated and conflicts resolved in order to accommodate these priorities. Define phases of partial capability during development or to be delivered if the entire program is not to be developed or delivered all at once.

4.4 Performance Requirements

4.4.1 Reviews and Approvals

Identify all required reviews, the content of such reviews, the makeup of the review board, the convening authority, and the action of the board; state the degree to which board approval, or other authority, is required in order to continue development.

4.4.2 Change Procedures

Describe procedures to be followed by the developers, users, or others in soliciting or effecting changes to the SRD if not already covered by Section 4.1.1.

4.4.3 Performance Measures

State criteria, in order of importance or by weight, by which the performance of the developers will be judged satisfactory, such as (1) effective control of cost, performance, and schedules; (2) timeliness in responding to new guidelines or events; (3) achievement of technical goals; (4) sensitivity to environment, including rules and policies; (5) conformance to standards requirements; (6) maintenance of adequate progress visibility; etc.

4.5 Documentation and Reporting Requirements

Identify the types of reports and documentation to be produced, and establish specifications governing the level of detail, format, medium,

quality, and mode of delivery. Include requirements for interim progress and QA reports, as well as planning, design, coding, testing, and maintenance documentation, as appropriate.

5. FUNCTIONAL REQUIREMENTS

(This section documents the overall functional characteristics required of the program, to a level of detail sufficient to allow the architectural design to proceed with adequate assurance to management that what was approved will actually be done.)

Develop the program I/O and processing requirements in hierarchic levels of detail. Use graphic aids, such as information flow graphs, block diagrams, or procedural flow graphs, and narrate the requirements concisely, but clearly and adequately.

5.1 Functional Overview

Describe the top-level requirement details of the operational software. Typical coverage at this point might address the structure, data flow, functional interfaces, major operating modes, utility factors, security/protection, planned later modifications or extensions, etc. If graphics are used, support these with narrative explanations. Although such requirements may be general in nature, they should, nevertheless, be described in complete sentences (e.g., "Decompose the telemetry stream into frames" is preferred over "process telemetry").

5.2 User Requirements

State any specific procedural, format, or information content requirements on the software (program plus documentation) that relate to the functional behavior as viewed by the user(s).

5.3 Operational Requirements

State any specific procedural, format, control, or information content requirements on the software (program plus documentation) that relate to the functional behavior as viewed by the operator(s).

5.4 Input, Processing, and Output Requirements

Present a hierarchic development of input, processing, and output requirements (using HIPO or other diagrams if needed). At each level in the hierarchy, present the following information (either structured or integrated narrative) for each functional requirement being described:

Input. Present a general description of the input parameters, special data, necessary formats, initial conditions, and special controls that

apply to the requirement being defined. Include, where pertinent, requirements for units of measure, limits and/or ranges of acceptability, accuracy/precision, frequency of arrival, etc.

Processing. Describe the processing required on the input data, including requirements for transformations, sequencing, logical concepts, timing, reductions, internal program checks, accuracies, tolerances, data manipulations, throughput rate, and control options, as appropriate, in a logical presentation sequence. Include equations to be solved if necessary. If the program requires a particular sequence of operations and/or a significant decision process, include their descriptions (e.g., as flow chart and/or decision table, plus narrative).

Output. Describe output requirements on display, storage, transmission, control, or other data, including units of measure, accuracy/precision, frequency of output, media, etc., as appropriate.

Interfaces. Describe, as needed, any additional detailed functional relationships of the interfaces of the program with other programs, the system, or other processing requirements herein described. If a system/subsystem interface specification has been produced, it may be referenced as the controlling document.

Diagnostics. Specify the requirements for any diagnostics, error detection, and recovery that must be designed into the function being described. (In general, other program diagnostics will also be designed into the program during implementation; however, those diagnostic features shall be required to include any diagnostics prescribed in this section.)

5.5 Data Base Requirements

Describe in general and quantitative terms the requirements for the data base(s) to be produced or used by the program. Include parameter requirements that affect the overall design of the program (or system of programs in which this program is a part), emphasizing data characteristics and data relationships, and including ranges, units of measure, accuracy/precision, etc., where applicable. Identify constraints that could prove critical during design and implementation. Describe, as appropriate, requirements for data collection, conversion, and distribution.

6. ACCEPTANCE AND EVALUATION CRITERIA

6.1 Competing Characteristics

Include a list of factors or features that compete for development resources and order these linearly, or in a matrix, according to their

importance in meeting the program technical requirements, such as (1) program size, (2) execution speed, (3) cost to develop, (4) time to develop, (5) vulnerability to operator or input error, (6) maintainability, (7) growth potential, (8) documentation readability, (9) portability or machine independence, and (10) cost to operate. Identify any known special circumstances that may tend to negate or reorder these priorities. State how major conflicts in priorities are to be resolved.

6.2 Acceptance Requirements and Criteria

6.2.1 Demonstration Configuration

State requirements relating to the location, configuration, conduct and review of acceptance tests.

6.2.2 Criteria for Acceptance

Provide a list of criteria or conditions defining when the software development task is complete, and, therefore, when the formal transfer of responsibility to operational and user organizations can take place.

6.2.3 Testing Requirements and Criteria

State specific requirements on the form and extent of tests that demonstrate the acceptability of the software, or reference such documents as contain this information. Include, as appropriate, requirements for demonstration of control and data I/O interfaces, logical conditions, processing, and diagnostic functions. Specify performance criteria by which test results will be judged to have demonstrated acceptable behavior.

6.2.4 Test Result Documentation

Reference applicable documents that specify the format, content, quality, and level of detail required in assembling the test data into a coherent record of test results; otherwise, provide these requirements in this section.

6.3 Quality Assurance Requirements

Reference applicable documents that specify, or identify in this section, the QA organization and its level of involvement in the software deliverables. Include, as appropriate, requirements for inspections of workmanship, configuration control, material custodianship, quality control, software (documentation) audits, etc. Provide criteria that specify the extent to which QA reports concerning the adequacy of the delivered software are a binding condition for delivery.

7. APPENDICES

Appendices may include, but are not limited to, explanatory material and requirements of an auxiliary nature, inserted directly or bound separately

for convenience. The following suggested topics are typical. Appendices may be designated as "Appendix A," etc., if desired, rather than by the Dewey-decimal system given here.

7.1 Glossary

Provide an alphabetical listing of terms, symbols, mnemonics, acronyms, etc., used in the body of the SRD having a special, not widely known use.

7.2 References

Provide a list of all source documents, standards, procedures, and references cited in the SRD body (or elsewhere in the appendices). Give a short description of the information referred to in each, if not evident from the title.

7.3 Analyses

Provide analyses of program requirements as appropriate to support or clarify functional requirements stated elsewhere in the SRD.

7.4 Formats and Auxiliary Tables

If detailed format requirements have been established by this SRD, and if these are more appropriately accommodated in an appendix rather than in the SRD proper, then insert such information in this section. Similarly, insert any requirements appended in tabular form in this section (or, perhaps, as separate appendices).

7.5 Typical Run Examples

In cases where illustrations of requirements take the form of examples, to which the executing program is to conform, and when these requirements are better appended than inserted in-line in the SRD proper, then insert such examples in this section.

APPENDIX D

SOFTWARE DEFINITION DOCUMENT OUTLINE

This appendix contains a detailed outline for the assemblage of program design definitions into a document for management visibility and approval. On completion, the document defines the program functional and internal architecture, costs, schedule, development plan and related matters to that extent which permits a competent technical and management review of the software to be delivered.

Figure D-1 is a top-level view of the document organization; greater hierarchic detail is provided in the detailed outline that follows. The detailed outline also contains, along with each topical heading, a description of the material to be inserted at that point.

Much of the material cited herein for inclusion can perhaps be satisfied by references to suitable documentation elsewhere, or by attachment to the SDD. Such practice should be encouraged whenever the reference documents are stable, or are under some acceptable change control mechanism, or else when any instabilities or changes in those references are not apt to affect the management information given in the SDD. Technical details at this point are primarily important only to the extent that they impact management decisions.

The principal guideline for the SDD content is that it includes only that amount of technical detail needed early in the development, which defines management resource requirements, the program general architecture, refined costs, and refined schedules. The SDD is not generally maintained after program delivery.

See Chapter 11 for more specific and detailed rules for completion of the SDD.

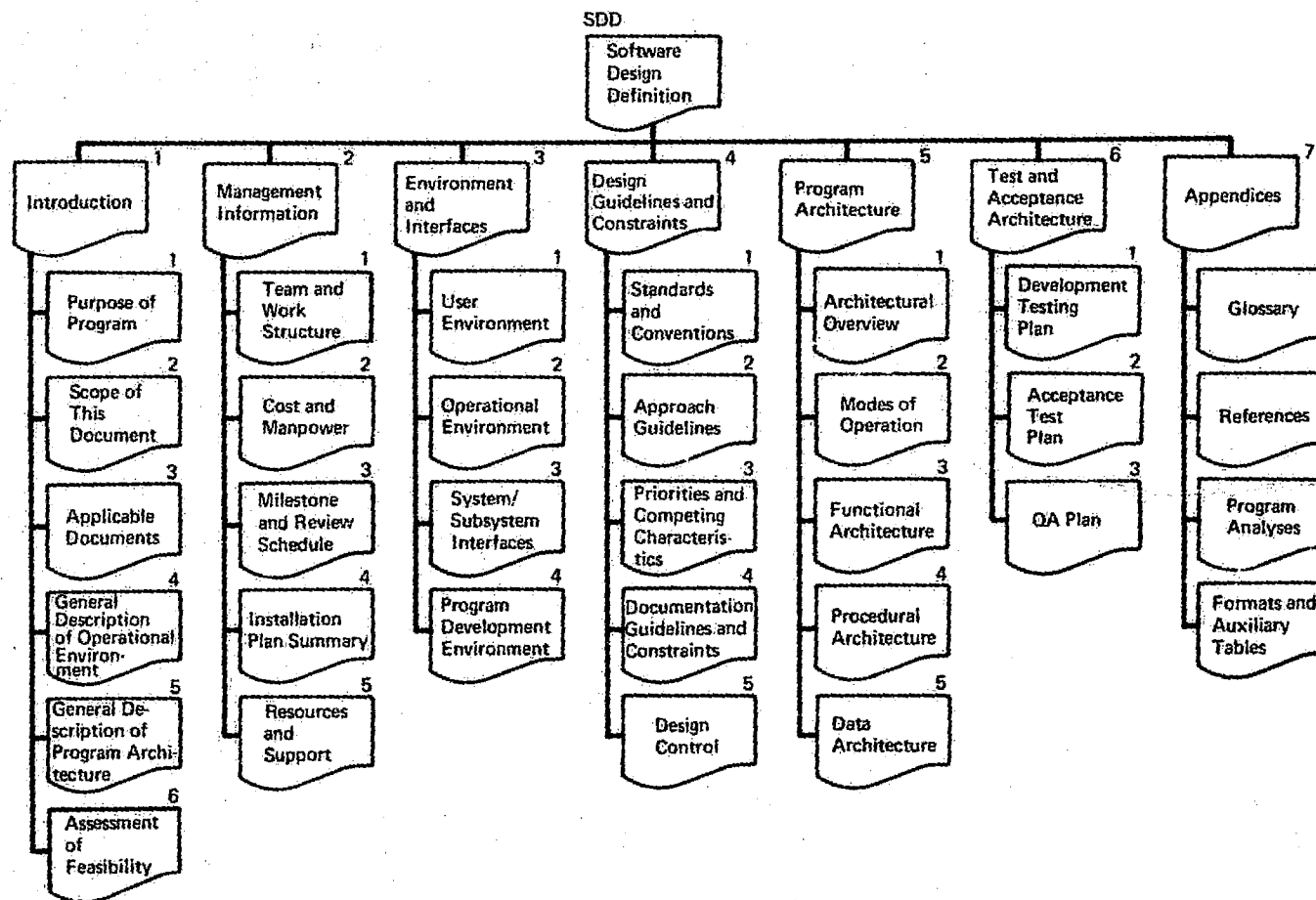


Figure D-1. A hierarchic outline for the SDD for describing the program architectural design

SOFTWARE DEFINITION DOCUMENT

Detailed Table of Contents

FRONT MATTER

Title Page. Provide a title page containing (1) document number; (2) SOFTWARE DEFINITION DOCUMENT; (3) title of the program, subsystem, and system; (4) the publication date; (5) signature approval block appropriate for the management authority level required; and (6) releasing organization. Approval signatures will be applied to continue detailed software development.

Abstract. Give a brief abstract that summarizes the material contained in the SDD.

Change Control Information. Provide a brief statement that specifies the level of change control to be exercised on the SDD during the time of preparation. On completion, revise this statement to reflect the proper post-signature change control policy and procedure.

Distribution Information. Provide a distribution list of all parties with need-to-know status of the SDD who are to receive copies. Provide a statement that tells how additional copies may be obtained.

Table of Contents. Provide a detailed table of contents for the completed SDD, which lists section number, title, and page of every item with a heading. This table is probably the last item supplied to the SDD prior to completion.

TEXT OF DESIGN DEFINITION

1. INTRODUCTION

1.1 Purpose of the Program

Provide a brief statement of the purpose of the program and of this SDD.

1.2 Scope of this Document

Provide a brief statement that defines the scope of material contained in this SDD, including a brief statement that scopes each major section of the document, if necessary.

1.3 Applicable Documents

Identify all controlling documents to which this SDD responds.

1.4 General Description of the Operational Environment

Provide an orientation for the reader that shows the software imbedded in its operational environment (both hardware and software). A dataflow block diagram with explanatory narrative is useful here. Identify the general mode of operation of the program, such as "stand-alone, interrupt-driven, real-time program on dedicated XXXX minicomputer operating under OSOSOS operating system with a 64K-word core and two 10M-byte disks." Identify any other general system constraints imposed by requirements or are otherwise critical to the design. Defer details, however, to Section 3. The emphasis at this point is on communicating an understanding among readers concerning the overall operational interfaces, and not on the detailed computer and peripheral interfacing aspects.

1.5 General Description of the Program Architecture

Provide an overview of the program function, response to requirements, nature of the problem and type of data input, processed, and transmitted, etc. Identify the type of program, such as: (1) real-time, interactive, or batch; (2) computational or data manipulation; and (3) developmental or operational. The descriptions contained in the SRD may be referenced and may provide a great portion of the level of understanding needed at this point. However, any new general information not in the SRD should be given. A data-flow or diagram, tier chart, or other form of high-level program block diagram which displays the program architecture may be useful to illustrate the program and its major mode of operation. Defer details to Section 5.

1.6 Assessment of Feasibility

Summarize the results of program analyses, cost and schedule considerations, availability of personnel, and other factors (such as, perhaps, salability, profit, organizational goals, etc.), which impact the development of this piece of software. If alternate plans are available, being developed under similar criteria or for special contingencies, then identify these alternatives and their viability. Discuss problem areas and areas of concern resulting from architectural study.

2. MANAGEMENT GUIDELINES

This section is a summary of the project resources, work schedule, and plans for implementing the requirements given in the SRD. The detail includes estimates for costs and schedule based on an analysis of the individual functions to be provided, the estimated time to design, code, test, and document each such function and configuration to be delivered, and the resources available to perform the task. A set of characteristics that compete for machine and management resources is prioritized as a guide toward implementation (Section 4.3), and it is upon this ordered priority that the given resource estimates are valid. The schedule and work plan display how the management resources are organized to accommodate these priorities, so that, in the unlikely events of schedule or cost overruns, the highest priority items at least will have been provided.

2.1 Team and Work Breakdown Structure

Identify the manpower allocations for development and show a breakdown of work into tasks. Identify personnel assigned to tasks by name when possible. Identify project management, supervision, design, coding, documentation, testing, QA, review, secretariat, and liaison functions. Based on work identified in each of the task areas, available personnel, planned work phasing, and the project team structure, provide a manpower profile for the software production, testing, documentation, and delivery. Include favorable and adverse tolerance figures for these estimates and state the basis for such estimates. In particular, state the assumed individual productivity (say, as lines of code per day per individual) used in estimating costs and the schedule of the following sections.

Provide an analysis based on the Work Breakdown Structure of manpower loading, flow of expenditures for computer facilities and services, and other factors that influence the level of support required for the software development. Provide a definition of the work priorities and indicate how the design and implementation process accommodates the priority ranking.

2.2 Cost and Manpower

Give the projected or allocated dollar and manpower costs to complete the program through documentation, testing, and delivery.

2.3 Milestone and Review Schedule

Present a more detailed, sharp-milestone refinement of the schedule given in the SRD, and point out any major differences, problems, reservations, or qualifications.

2.4 Installation Plan Summary

Summarize the plans for implementing the software and integrating it into its system and operational environments. Describe any negotiations or agreements necessary to end the development activity and deliver the program into operation.

2.5 Resources and Support

State the planned resources and also support other than dollars and manpower that will be required to develop the software, insofar as these are drivers for cost, schedule, manpower, or program architecture. Include availability of critical facilities, subcontract negotiations, etc., as appropriate. Identify deficiencies, either incompletely resolved or undefined, as *liens* only if these are currently a problem or if they typically would not be resolved later, prior to projected needs during actual work phasing.

3. ENVIRONMENT AND INTERFACES

This section expands upon the overview presented as Section 1.3. Its function is to define the scope of the task, not to define the system, environment, and interfaces in detail.

3.1 User Environment

Describe the user interfaces with the program and the functions that user interactions invoke within the program. Use references to user manuals, the SRD, and other such materials that describe those aspects. Include only that information which impacts either the program architecture or the management resources needed for implementation. Leave other details for the SSD and the User's Manual(s).

3.2 Operational Environment

3.2.1 Operational and Operator Interfaces

Describe operational and operator interfaces with the software being developed to that level of detail at which impacts to management resources and major technical decisions are typically felt. Use references to the SRD, operator manual(s), technical manuals, liaison personnel, etc., to obtain the level of detail and information deemed proper.

3.2.2 Operations/Maintenance Plans

Describe the measures to be used or developed for maintaining or operating the program prior to delivery, and indicate the suitability of these measures as preliminary operational methods after delivery. Identify the post-delivery operations/maintenance responsibility.

3.3 System/Subsystem Interfaces

Outline both the hardware and software aspects of the system and/or subsystem that are needed to define the program architecture or else are projected to influence the program architecture significantly if not identified and resolved early. Indicate maximum available core, segmentation constraints, system-imposed timing constraints, the use of existing or standard subroutines, etc., as appropriate. Reference existing hardware and software technical manuals, system technical requirements documents, etc., if appropriate.

3.4 Program Development Environment

Describe and discuss any aspects of the environment in which the software is being written, designed, coded, tested, etc., that will contribute to adverse or favorable deviations from the plan of this SDD, should certain identified contingencies be or not be realized. For large programs that require a work breakdown structure, describe the functional interfaces which are to be maintained during program development. This section is generally only warranted when problems are projected to arise due to the development environment.

4. DESIGN GUIDELINES AND CONSTRAINTS

This section consists of material that defines and constrains the development disciplines and conventions used by the project.

4.1 Standards and Conventions

Define in this section those special standards and conventions that apply to this SDD. Additionally, identify or describe any special standards or conventions necessary for developing the software, but only if these are a major consideration in establishing the costs, manpower, schedule, or program architecture. Established standards and conventions may be cited by reference.

4.2 Approach Guidelines

Define any special methodology to be used to develop, code, or test the software, and state any special approach or design philosophy toward solving the software design problems, insofar as these are instrumental in determining costs, manpower, schedule, or program architecture. Standard methodologies or approaches will have been covered by 4.1, above.

4.3 Priorities and Competing Characteristics

List the significant technical and management factors that compete for costs, schedule, and program and documentation quality. Prioritize these so as to fulfill the design philosophy, documentation plans, and SRD

requirements. Include such considerations as program size, execution speed, vulnerability to operator error, maintainability, growth capability or extension, portability of design, readability of documentation, level of documentation, vulnerability to system errors, etc., but only include such factors if they are deemed to impact costs and schedules.

4.4 Documentation Guidelines and Constraints

Identify the documents to be produced, the purpose, type, and level of quality of each, the intended reader, the projected life cycle, and the allocated resources that have been planned for each. Identify any documentation factors that need resolution and that will significantly impact management resources. Include, as appropriate, project logs, reports, notebooks, etc., as well as deliverables. Name individuals responsible for each.

4.5 Design Control

State summarily the level of design control exercised within the project, and state the impact of this philosophy on technical and management resources. Describe the methods and disciplines to be imposed or developed to assure configuration control and management of the evolving software.

5. PROGRAM ARCHITECTURE

This section reveals the top-level program functional and modular hierarchy of algorithmic design structure. Its function in the SDD is to provide credibility for cost/schedule/manpower estimates and allocations, and a basis for reviewing the architectural design that will be used in the SSD.

This section should demonstrate that the program architecture satisfies the following criteria: (1) adequacy to fulfill the program requirements, (2) adequacy for continuing the design, (3) adequacy for coding the design from the top down on a module basis, (4) adherence to program development standards, and (5) adequacy for use of the SSD as the principal sustaining document after software delivery.

5.1 Architectural Overview

Describe and discuss, in summary form, the overall program organization, both functionally (external characteristics) and algorithmically (internal characteristics). Only the major functions, algorithms, data structures, and inputs/outputs that define the program architecture need to be addressed at this point. Such descriptions may take the form of modular data flow descriptions within the system or subsystem, or within the program; they may take the form of a high-level tier-chart with explanatory

narrative; they may take the form of structured-control flowcharts and data-structure diagrams (always with explanatory narrative) to show the program algorithmic composition; and, for complex control-logic situations, they may take the form of modular decision tables. Indicate, based on the architectural study, the approximate core occupancy, segmentation requirements, timing constraints, etc.

5.2 Modes of Operation

Identify the major operating modes (or ways of operating the program, as selected by control data inputs) and the different software configurations (if more than one exists). Discuss summarily the behavior of the program in each mode or configuration, the stimulus that causes transitions between modes, and the rationale by which control data selections are available at each transition.

5.3 Functional Architecture

Describe the functional (or externally observable) behavior of the program, including the detection of, and recovery from, system failure and input errors. Use hierarchic refinement of detail to that level which defines the functional architecture of the program sufficient for a high-level design review. Identify each function performed with a requirement in the SRD, or else justify each such function within some set of valid design criteria.

The emphasis in this section is on the main sources and characteristics of the input, the main types and characteristics of output data, and the intermediate flow through the transformational processes involved, for functional definition and requirement interpretation, rather than for detailed procedure/program control, the subject of Section 5.4.

5.4 Procedural Architecture

Provide a level-1 flowchart (or equivalent) and narrative that discusses the entire program algorithm and each functional step. Correlate operative modes and external functional behavior elements with functions or partial functions associated with each procedural step. Expand functions to be hierarchically refined ("striped" or "named" modules) at succeeding levels, using the standards given in Chapter 12, to that level of refinement required to give credence to cost/schedule/manpower estimates, to define the program structure, and to illustrate the design and documentation standards that will appear in the SSD. Correlate functions that operate on major data bases and internal data structures with material in Section 5.5.

If it is an appropriate design consideration, state what language(s) will be used for coding and why.

5.5 Data Architecture

Identify and describe the major data bases accessed or created by the program being developed and identify and describe major data structures used within the program. Describe only the high-level design considerations of such data, hierarchically refined, as dictated by the functional and procedural descriptions in Sections 5.3 and 5.4. In some cases, the baseline design core map may serve to illustrate memory requirements or constraints. Include, as appropriate, considerations for data collection, conversion, and distribution, insofar as these are design concerns that impact cost and schedule.

6. TEST AND ACCEPTANCE ARCHITECTURE

6.1 Development Testing Plan

State procedural guidelines to be used by the development team in evaluating program correctness prior to acceptance testing. Discuss the projected adequacy of such methods to reduce program errors prior to verification (acceptance) testing.

6.2 Acceptance Test Plan

Identify plans, procedures, configurations, support, and personnel deemed necessary for acceptance testing, both in response to the SRD and as prompted by considerations recognized in the architectural design phase. Identify testing and reviewing personnel.

6.3 Quality Assurance Plan

Identify the level and type of involvement by QA personnel during the project prior to delivery. Identify specific functions and services that QA personnel are sought to provide (e.g., audits of performance vs. requirements, code vs. requirements, code vs. flowcharts, testing the program, etc.); and state what materials will be provided for QA action.

7. APPENDICES

Appendices may include, but are not limited to, explanatory material of an auxiliary nature, inserted directly or bound separately for convenience. The following topics are typical. Appendices may be designated as "Appendix A," etc., if desired, rather than by the Dewey-decimal system given here.

7.1 Glossary

Provide a glossary of all acronyms or mnemonics and all frequently used, unfamiliar terms used in the SDD. Give short definitions of each, and a reference, when appropriate, to a fuller definition in the text.

7.2 References

Provide a list of all source documents, standards, procedures, text books, etc., used as reference material in the SDD text. Give a short description of the information referred to in each, if not evident from the title.

7.3 Program Analyses

Include, as appropriate, any program analyses that may tend to support, illustrate, clarify, or, in some cases, refine the material or decisions contained in the SDD. For example, a decision to code the design in assembly language may be supported by a timing analysis of critical, high-priority external functions that can only be accommodated in assembly code. The results of coding and testing the architectural portion of the design (using dummy stubs) may be offered in evidence of correctness, etc.

7.4 Formats and Auxiliary Tables

If formats or other auxiliary tables have been established by this SDD and are appropriate appendix material, include these in this section.

MISSING PAGE BLANK NOT

APPENDIX E

SOFTWARE SPECIFICATION DOCUMENT OUTLINE

This appendix contains a detailed outline for the accumulation of all program specifications into a single document. The items listed are not meant to be filled out in the order listed as a function of time. Rather, different sections should be worked on as they are appropriate to the program development effort. The outline given then organizes that material into logically related sections for readability. On completion, the document forms the as-built specification of the program, and serves as the principal maintenance document.

Figure E-1 is a top-level view of the document organization; greater hierarchic detail is provided in the detailed outline that follows. The outline contains, after each topical heading, a description of the type of material to be inserted at that point.

The level of detail in the SSD may vary according to the documentation needs of the program. The topics given are meant to serve as a documentation checklist. In full, the topics constitute Class A documentation detail.

Much of the material cited herein for inclusion can perhaps be satisfied by references to suitable documentation elsewhere, or by attachment to this SSD. Such practice should be encouraged whenever the reference documents are either stable or under the same change control mechanism as this SSD, or else when any instabilities or changes in those references are not apt to affect the "as-built" character of this specification.

Chapters 11 and 12 contain more specific and detailed information relative to completion of the SSD.

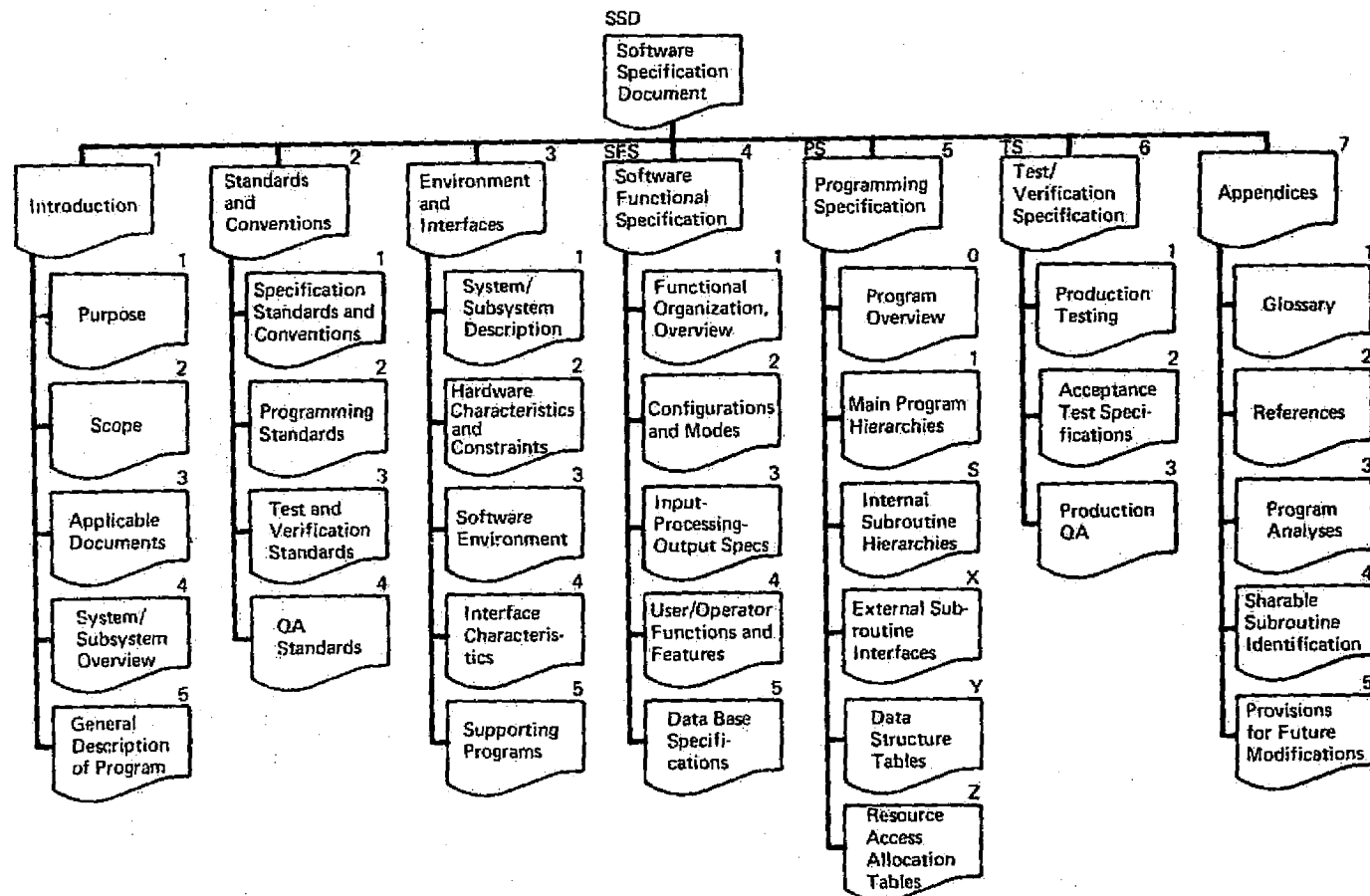


Figure E-1. Graphical outline of the SSD

SOFTWARE SPECIFICATION DOCUMENT

Detailed Table of Contents

FRONT MATTER

Title Page. Provide a title page containing (1) document number; (2) SOFTWARE SPECIFICATION DOCUMENT; (3) program, subsystem, and system titles; (4) the publication date; (5) a signature approval block appropriate for the management authority level required; and (6) releasing organization. Signatures are to be supplied only at SSD completion. The date reflects the time of last change to any item in the SSD.

Abstract. Give a brief abstract that summarizes the program specification.

Change Control Information. Provide a statement that specifies the current level of change control authority during production, and procedures for updates. On completion, revise this information to reflect the proper post-signature change control policy and procedure.

Distribution List. Provide a distribution list of all parties with need-to-know status of the final complete SSD who are to receive copies.

Distribution Information. Provide a statement that tells how additional copies may be obtained.

Table of Contents. Provide a detailed table of contents for the SSD, which lists section number, title, and page of every item with a heading. This item is probably the last supplied to the SSD.

TEXT OF SPECIFICATION

1. INTRODUCTION

1.1 Purpose of the Program

Provide a brief statement of the purpose of the program and of this specification.

1.2 Scope of Applicability

Describe the scope of the document, including brief statements about each major section of the document, if necessary.

1.3 Applicable Documents

List controlling and source documents that apply to this specification, such as the SRD, SOM, user manuals, external functional or programming specifications, etc.

1.4 System/Subsystem Overview

Identify the general system, subsystem, and environment (hardware and software) in which the program operates (a block diagram with narrative is useful here). Also, state the general system/subsystem operating mode, such as "interrupt-driven, real-time, on dedicated MODCOMP operating under MAX/III with 16K-word core and 5M-byte disk." Identify any other general system constraints imposed by requirements that influenced design, such as allocated core size, system-imposed timing constraints, etc. Leave details describing the system/subsystem to Section 3.

1.5 General Description of the Program

Provide a general functional description of the program, including the nature of the problem and the type of data generated, processed, or transmitted. Identify the type of program, such as: (1) real-time, interactive, or batch; (2) computational or data manipulation; (3) developmental or operational. If appropriate, describe the method of solution.

2. STANDARDS AND CONVENTIONS

This section describes standards and conventions used in this SSD and in programming, to describe both the internal and external characteristics of the program.

2.1 Specification Standards and Conventions

2.1.1 Applicable Documentation Standards

Identify all existing documentation standards that are used to document the program as appropriate references to Section 7.2 of the SSD.

2.1.2 Exceptions to Specified Documentation Standards

Identify all exceptions to the standards specified in 2.1.1, above, and provide alternate standards in their stead, if required.

2.1.3 Special Documentation Standards

Describe any special standards, such as formats for graphics (e.g., Petri diagrams) or descriptions of data structures (e.g., by a PASCAL-like syntax),

that are not covered by standards above. Define special terms or symbols, especially those used in non-standard ways, or in ways that are likely to be misunderstood by envisioned readers.

2.2 Programming Standards

This section describes standards and conventions used to implement the program and to document the program internal characteristics.

2.2.1 Applicable Policy and Procedure Documents

Identify appropriate policy and procedure documents that govern the program implementation philosophy, discipline, or approach by appropriate reference to Section 7.2 of the SSD.

2.2.2 Exceptions to Established Policies and/or Procedures

Identify extensions, modifications, additions, or other exceptions to the documents listed in 2.2.1, above, if needed.

2.2.3 Special Policies and Procedures

Describe any special policies applied toward the development of this program not covered by Sections 2.2.1 and 2.2.2, above.

2.2.4 Applicable Programming Standards

Cite any existing programming standards used by appropriate reference to Section 7.2 of the SSD.

2.2.5 Exceptions to Specified Programming Standards

Identify all exceptions to cited standards in 2.2.1, above, and state alternate standards used in their stead, if required.

2.2.6 Special Programming Standards

Describe any special standards adopted for this program, such as: (1) register definitions and usage; (2) naming of variables or program labels; (3) assumed form of subroutine calling sequences and other procedural linkages; (4) use of global macros; (5) naming of compile-time constants and parameters; (6) method used to prevent inadvertent misuse of program resources, e.g., as duplication of variable, file, and label names; (7) method used for arbitration of requests for use of shared resources; (8) method used to assure that deadlocks cannot occur; etc.

2.2.7 Programming Language(s)

Indicate the specific language(s) used and any language extensions allowed, including macros, when applicable.

2.3 Test and Verification Standards

2.3.1 Applicable Test and Verification Standards Documents

Identify all standards documents that are applicable to the establishment of program correctness and to the demonstration of acceptance criteria by appropriate reference to Section 7.2 of the SSD.

2.3.2 Exceptions to Specified Test and Verification Standards

Identify all exceptions to the standards specified in 2.3.1, above, and state alternate standards in their stead, if required.

2.3.3 Special Test and Verification Standards

Describe any special testing standards, such as method of assessing correctness, reporting and archiving test data, benchmark tests, etc.

2.4 Quality Assurance Standards

2.4.1 Applicable QA Standards

Identify all existing QA standards documents that are applicable toward inspection for transfer to operation by reference to Section 7.2 of the SSD.

2.4.2 Exceptions to Specified QA Standards

Identify all exceptions to the standards specified in 2.4.1, above, and state alternate standards in their stead, if required.

2.4.3 Special QA Standards

Describe any special standards, such as rules for cross-auditing narrative, flowcharts, code, and test results, or for auditing narrative format, flowchart conventions, indentations or annotation of code, decision table formats, etc.

3. ENVIRONMENT AND INTERFACES

3.1 System/Subsystem Description

3.1.1 Applicable System Documents

Reference those source documents and manuals listed in 7.2 that describe the general interface characteristics of the system or subsystem in which the program resides.

3.1.2 General Description of System/Subsystem Environment and Interfaces

Identify the system/subsystem interfaces, and describe the program and system environment in an introductory manner, for further detailing in

appropriate subsections of this section. Use the narrative in this subsection to expose the reader to the material contained in the coming subsections, and to guide him verbally toward a general understanding of the material.

3.2 Hardware Characteristics and Constraints

Identify and describe the pertinent hardware functionally. However, defer detailed descriptions of hardware items, using references to specifications documents and technical manuals listed in Section 7.2 of the SSD whenever possible, if not covered in 3.1.1, above. Otherwise, supply such information in Sections 3.2.1-3.2.7, below, as appropriate.

3.2.1 General Description of the Hardware

Provide a narrative description of the pertinent aspects of the hardware in which the program operates, including functional diagrams, as appropriate.

3.2.2 Machine Configuration

Include a block diagram of the main frame and peripheral configuration, if appropriate.

3.2.3 Main-Frame Characteristics

Identify those main-frame characteristics that are pertinent to the software design, such as: (1) read-only memory, (2) floating point hardware, (3) allocated core size, etc.

3.2.4 Peripheral Characteristics

Identify the type and quantity of standard peripherals accessed by the program. Describe any peripheral characteristics that are pertinent to the software design.

3.2.5 User/Operator Control Interface Characteristics

3.2.5.1 Control Input Devices

Identify all input devices utilized for control of the program, such as: (1) keyboard, (2) card reader, (3) control panel, (4) special devices.

3.2.5.2 Control Monitor Devices

Identify all output devices utilized for responding to controls, such as: (1) typewriter, (2) CRT/hard copy, (3) line printer, (4) control panel, (5) special devices.

3.2.5.3 Control Device Characteristics

Identify any special characteristics of control devices, such as: (1) ASCII, EBCDIC, or Fieldata character codes; (2) line rates; (3) half duplex or full

282 Appendix E

duplex; (4) system-imposed characteristics, e.g., data transfer mode by character or by block.

3.2.6 User/Operator Data Interface Characteristics

3.2.6.1 Data Input Devices

Identify all output devices utilized by the user/operator for data input to the program, such as: (1) keyboard, (2) card reader, (3) control panel, (4) special devices.

3.2.6.2 Data Output Devices

Identify all output devices utilized for data output from the program to the user/operator, such as (1) typewriter, (2) CRT/hard copy, (3) line printer, (4) card punch, (5) plotter, (6) special devices.

3.2.6.3 Data Input/Output Device Characteristics

Identify any special characteristics of user/operator data input/output devices, such as: (1) ASCII, EBCDIC, Fieldata character codes; (2) line rates; (3) half duplex or full duplex; (4) automatic control codes or other non-printing control characters.

3.2.7 Data Storage Device Interface Characteristics

3.2.7.1 Data Storage Input/Output Devices

Identify all mass-storage devices utilized for data or control inputs and outputs to the program, such as: (1) disk/drum, (2) magnetic tapes, (3) paper or cassette tape, (4) other mass memory devices.

3.2.7.2 Data Storage Device Characteristics

Identify any special characteristics of mass-storage devices, such as: (1) minimum sector size of disk/drum, (2) number of tape recorder tracks, (3) tape blocking factor, (4) ASCII, EBCDIC, or Fieldata character codes.

3.3 Software Environment

Provide a narrative description of the system software that imposes constraints on the program design. Identify the compiler(s) or assembler(s) used, by version if necessary, and the mode of operation, such as batch, multiprogram interactive, background batch, etc., if appropriate. Do not enter information that does not have a direct bearing on the design. If necessary, however, supply such information in Sections 3.3.1-3.3.3, below.

3.3.1 General Description of System Software

Describe the general organization of system software, with diagrams as appropriate.

3.3.2 Interaction With the Operating System

Describe how the program interacts with the operating system, and describe any special features of the operating system to accommodate this design.

3.3.3 Interaction With Other Programs

Describe how the program interacts with other programs, including library calls for standard subroutines, program linking/overlaying, and those operating in a multiprogramming environment, if not adequately described elsewhere in the SSD.

3.4 Interface Characteristics

3.4.1 Operating System Interfaces

Provide a description of operating system/program interfaces, or give a reference to such material elsewhere in the SSD, or to a document listed in Section 7.2.

3.4.2 Interfaces With Other Programs

Provide descriptions of interfaces with other programs or give a reference to such material elsewhere in the SSD, or to a document listed in Section 7.2. Identify programs that provide data to or receive data from the program. Identify any special characteristics related to the data transfer between programs. Define the parameters for data messages and the characteristics of data control parameters associated with the data.

3.4.3 External Hardware Interfaces

Identify all non-standard or specialized input/output devices. Define the pertinent characteristics of these devices, such as: (1) control parameters, (2) response parameters, (3) interrupts, (4) interrupt priority, (5) timing constraints. If such information exists in a source document, it may be more appropriate to cite such a reference than to repeat that information here.

3.5 Supporting Programs

Describe or give a reference to any programs used for operational or developmental support of this program, if such descriptions are needed to understand the program specifications herein contained. In such cases, describe the interfaces and interactions with such programs.

4. SOFTWARE FUNCTIONAL SPECIFICATION

4.1 Functional Organization and Overview

Describe the overall functional behavior of the program, the principal modes of operation, the different software configurations (if any), and major

data flows. Cite applicable requirements documents and user/operator manuals by appropriate references to Section 7.2 of the SSD. The object of this section is to prepare the reader for the detailed functional specifications to follow. In this overview, discuss the general philosophy concerning detection of and recovery from system failure and input errors; if needed, attach these as appropriate Sections 4.1.i and 4.1.j (*i* and *j* are appropriate numerals within 4.1), as detailed below.

4.1.i Detection of and Recovery From System Failure

Describe the technique to be used to detect and recover from system failure, such as: (1) re-run program, (2) writing of checkpoint recovery file, (3) storing data on disk and/or tape at specified intervals, (4) backup system on line, etc.

4.1.j Detection of and Response to Data Input Errors

Describe the extent to which input data is to be verified before processing, and the verification method, such as: (1) checksums on data records or files, (2) error detection coding and retransmission, (3) message to operator for manual verification, etc.

4.2 Detailed Software Configurations and Modes of Operation

Identify each of the different software configurations (essentially different programs built or linked together as a unit). For each configuration, identify the various operational modes (program functions that change as a result of the operational state or control data). Describe events, conditions, or computations that cause transitions between modes. Use hierarchic refinement if modes have functional submodes that need description. Name and number each mode for input-processing-output descriptions in SSD Section 4.3. Use decision tables to express mode-transition logic and to identify major functions within modes as appropriate.

4.3 Input, Processing, and Output Specifications

This section contains hierarchically refined input-processing-output specifications and deals with end-to-end program functional characteristics. Each major Subsection 4.3.i, below, where *i* is a function or mode identifier, describes one of the program modes, and subsections within these refine the various aspects of the mode functional behavior.

4.3.i Function *i* Mode (or Mode *i*) Input-Processing-Output

Describe the inputs to this mode, the processing functions performed on the input, and resultant outputs and responses. Illustrate the mode inputs, functions, and outputs by a one-page data flow diagram, and explain the program behavior. Use decision tables to define responses to intricate logical conditions. Further information on generating such sections of the SSD is contained in Chapters 11 and 12.

C-1

4.4 User/Operator Functions and Special Features

Identify and summarize the user/operator interfaces and specify those functions performed by the user/operator. Alternatively, give a reference to the appropriate user/operator manual listed in Section 7.2 of the SSD. If a reference is given to such a manual in lieu of specifying operator functions here, then that manual becomes the operator functional specification, and should also be brought under the same change control mechanism as this specification.

4.4.1 Control/Response Message Parameters

For the input/output devices listed in Section 3.2.5, define control/response message parameters, such as: (1) structure and format (reference 7.7.1), (2) syntax, (3) message lengths and frequency, (4) device assignment codes, (5) special timing constraints, (6) special features such as access keys, passwords, or lockout keys, etc.

4.4.2 Data Input/Output Message Parameters

For the data input/output devices listed in Section 3.2.6, define user/operator data input/output message parameters, such as: (1) structure and format (reference 7.7.2), (2) syntax, (3) message lengths and frequency, (4) device assignment by message, (5) special considerations such as online vs. offline printing/display, (6) data units, (7) data ranges, etc.

4.4.3 Operational Environment and Support

Provide specifications for any operational support required for this program, and state any operational environmental characteristics on which the specification of this program is based, beyond normal or standard operational facilities, or give a reference to documents listed in 7.2 that provide such information.

4.4.3.1 Operator Functions

Specify other user/operator interactions required for operation of the program, and discuss any pertinent considerations for offline activities, such as transmittal or storage of output, operations log, etc., as appropriate.

4.4.3.2 Special Functions

Specify any other operational support functional requirements, such as criteria for running backup operational programs or re-entry of checkpoint data, etc.

4.5 Data Base Specifications

4.5.1 Interaction With External Data Bases/Files

Describe or give appropriate references to Section 3.3 of the SSD that state how the program interacts with data bases and files which are external to the program. Where appropriate, give data block formats, table formats, calling sequences, message formats, relationships between fields or records, units of measure, conversion formulas, etc.

4.5.2 Creation, Access, and Maintenance of External Data Bases/Files

Indicate whether or not data bases and files external to the program are created and/or maintained by this program. Discuss matters of privacy, security, and integrity of such data bases.

4.5.3 Applicable Data Base/File Documentation

Reference those source documents listed in Section 7.2 of the SSD that describe the pertinent data bases or data files, including those documents which describe how the data bases or files are created and/or maintained.

4.5.4 Data Base Descriptions

If there are no applicable documents referenced by Section 4.5.3, above, or if further specification is warranted in this SSD, provide that information in this section. Formats may be put in appendices, Section 7.7.3.

5. PROGRAMMING SPECIFICATION

5.0 Program Overview

Describe and discuss, in summary form, the solution method, the overall program organization, major internal data structures, and major algorithms that constitute the main program functions. Give design philosophy and rationale as appropriate for understanding, and give references to program analyses in Section 7.3, if any, which are pertinent to the overall program description.

The sections under this topical heading may describe relationships between the functional behavior and the executing modules of the program, i.e., an analysis of what modules do which functions. Other subsections may describe and depict data flow between executing modules or concurrent processes to illustrate how the program architecture accommodates the functional specification. Still other subsections may discuss the roles played by the various modules in each of the several program modes, or may show various data interface characteristics, when appropriate at this high-level design overview.

Insert Sections 5.0.i and 5.0.j, below, which discuss design philosophy and competing characteristics, at appropriate points *i* and *j* in the overview.

5.0.i Design Philosophy and Rationale

Include a brief description of the design philosophy, discipline, or approach taken in the program implementation, and give rationale as appropriate to explain such descriptions, insofar as such descriptions tend to explain why the design appears as it does.

5.0.j Ordered Set of Competing Characteristics

Describe the general order of priorities adopted to fulfill the design philosophy. Include an ordered list or table of factors that compete for resources, such as: (1) program size; (2) execution speed; (3) cost to implement; (4) time to implement; (5) vulnerability to operator error; (6) maintainability; (7) growth capability, including extension; (8) portability, or machine independence; (9) cost to operate; (10) readability of documentation; (11) concurrency of documentation; (12) vulnerability to system errors; etc.

5(1)* Main Program Detailed Design (Module 1)

Provide an algorithmic description of the top-level main program design, as described in Chapter 12. Hierarchic detailings of the main program into nested 1-page flowcharts with accompanying narratives (or equivalents) then follow as subsections.

5(n) Configuration n Detailed Design (Module n)

If the program has different identifiable compile configurations, show each such configuration beginning at its top-level flowchart and narrative, as above. Numeric identifiers can also be used for major program segments within a program, if desired.

5(S) Internal Subroutine Detailed Designs

5(Si) Subroutine i Detailed Design (Module Si)

Provide a detailed algorithmic description of the top-level design of each Subroutine *i*, as described in Chapter 12. Hierarchic detailings of each subroutine procedure into nested 1-page flowcharts with accompanying narratives (or equivalents) then follow as subsections.

* Numbers in parentheses refer to Dewey-decimal identifiers for flowcharts, data-structure definitions, or resource allocation requirements.

5(X) External Subroutine Interface Descriptions**5(Xi) External Subroutine i Interface Description**

Each section such as this in the SDD shall provide the necessary interface description for an external module called by this program being specified. Such description shall contain, but shall not be limited to: (1) the purpose and function of the subroutine; (2) the calling sequence; (3) all external programs and subroutines called; (4) common data areas; (5) operating system interface data; (6) mathematical equation, if appropriate; (7) execution speed and core usage, if relevant; (8) input/output; (9) restrictions for use; and (10) error messages. If suitable descriptions of this type are documented elsewhere, then a reference to such source material listed in 7.2 is sufficient.

5(Y) Data Structure Definition Tables**5(Yj) Data Structure Name j Definition Table**

Each section such as this shall describe common, global, or shared data structures. Such descriptions become the controlling interface definition for all accesses to the structure, and, when appropriate, identify those functions forming the level of access to the structure. Descriptions shall contain, but shall not be limited to: (1) mnemonic name and derivation of that name; (2) purpose and usage in the program; (3) structural description, including overall type, size, component breakdown, and graphic illustration(s); (4) substructure definitions, including for each field: type, size, position (if relevant), relations with other components, allowable operations, and constraints, such as ranges of value; (5) functions or operations that may manipulate the structure (or its members); (6) associated constants used to define structure parameters; (7) relationships with other structures; and (8) constraints on usage.

5(Z) Resource Access Allocation Tables**5(Zk) Resource k Access Design**

Each section such as this shall describe access requirements, protocols, methods for achieving mutually exclusive use, synchronization, etc., for a resource or sets of resources. These descriptions may take the form of hierarchic levels of access, described in layers of refined detail in subsections of each such section.

6. TEST AND VERIFICATION SPECIFICATIONS**6.1 Production Testing**

This section describes correctness testing by implementors.

6.1.1 Applicable Existing Production Test Procedures

Identify by appropriate reference to Section 7.2 of the SSD any existing and documented production testing procedures (other than standards in Section 2.3) used to support the validation of this program during development.

6.1.2 Exceptions to Existing Production Test Procedures

Identify all exceptions to existing test procedures referenced by Section 6.1.1, above, and state alternate procedures, when applicable.

6.1.3 Other Production Test Procedures

Define other production testing procedures used to support the verification of this program during development. Define criteria for test data selection, and state procedures for determining the validity of observed program responses to given test data. See Section 6.2.3, below, for suggested outline.

6.2 Acceptance Test Specifications

When appropriate, this section specifies tests that demonstrate the fulfillment of acceptance criteria.

6.2.1 Applicable Existing Acceptance Test Procedures

Identify by appropriate reference to Section 7.2 of the SSD any existing and documented acceptance test procedures (other than standards in Section 2.3) used for certifying the program.

6.2.2 Exceptions to Existing Acceptance Test Procedures

Identify all exceptions to existing test procedures referenced by Section 6.2.1, above, and state alternate procedures, when applicable.

6.2.3 Other Acceptance Test Procedures

This section defines, when appropriate in this document, other acceptance test procedures used to certify the program prior to operations transfer. Each subsection provides for the documentation of criteria for test data selection and procedures for determining the validity of observed program responses to given test data. Each description shall contain, but not be limited to: (1) an explanation of the test objectives, (2) test inputs (files, events, etc., or criteria for their selection), (3) test procedures, (4) support facilities required (used), (5) test conditions or constraints, (6) outputs to be achieved, and (7) method of output interpretation. A suggested list of considerations follows.

6.2.3.1 Verification of Program Control I/O Interfaces

Define tests and techniques to verify proper operation of the program with the specified user/operator control media (Section 3.2.5).

6.2.3.2 Verification of Program Data I/O Interfaces

Define tests and techniques to verify proper operation of the program with the specified user/operator data I/O media (Section 3.2.6) and data storage media (Section 3.2.7).

6.2.3.3 Verification of Data Processing Functions

Define tests to verify proper operation of the data processing functions of the program. Include test cases for normal, random, and abnormal (out-of-range or otherwise unexpected) data.

6.2.3.4 Verification of Logical Response and Sequence

Define tests and techniques to verify the proper logical response and sequencing of modes in response to control and/or data input parameters. Include test cases that verify responses to erroneous data or control parameters.

6.2.3.5 Verification of Diagnostic Functions

Define tests and techniques to verify proper operation of in-program diagnostics that aid the user/operator in detecting failures, correcting errors, or in determining the causes(s) of abnormal operating conditions.

6.2.3.6 Verification of Software Trap Action

Provide tests to verify proper operation of software traps used to prevent anomalous operation, such as: (1) file record errors; (2) software timers for endless loops; (3) out-of-range location address, including out-of-range of available memory; (4) impossible mathematical computations, e.g., divide by zero.

6.2.3.7 Verification of Error Response

Provide tests to verify that error conditions produce the proper responses, such as the correct error message.

6.3 QA Measures During Production

This section defines, when appropriate in this document, the level of activity and responsibilities of QA functions performed for certification and delivery into operations, insofar as these are not covered by QA standards contained in Section 2.4. Treated are such areas as (1) procedures for an audit of the complete software package for conformance to standards, as well as conformance of code to procedural specifications, procedural specifications to functional specifications, functional specification to requirements, test results to test specifications, and performance to requirements; (2) participation in design reviews; (3) standards enforcement; (4) configuration control; (5) discrepancy reporting; (6) change control; and (7) test conducting.

7. APPENDICES

Appendices may include, but are not limited to, explanatory material of an auxiliary nature, inserted directly or bound separately for convenience. The following topics are typical. Appendices may be designated as "Appendix A," etc., if desired, rather than by the Dewey-decimal system given here.

7.1 Glossary

List the names of the program, all subprograms and subroutines, and all variables and parameters used in program design, along with their mnemonic derivation. Also give short definitions of each and a reference to its detailed definition, when necessary. Include all acronyms, as well as frequently used, unfamiliar terms used in the program subsystem/system descriptions.

7.2 Listing of Source Documents and References

Provide a list of all source documents, standards, procedures, and reference material used for program design, implementation, and testing. Indicate the subject matter and purpose of each reference.

7.3 Program Analyses

Provide analyses of algorithms, program functions, timing profile diagrams, etc., as appropriate in support of design decisions made during implementation.

7.4 Sharable Subroutine Identification

Identify software items developed that have sharing potential, either as common software in other subsystems currently being developed or as candidates for use in future implementations.

7.5 Provisions for Future Modification

Identify those features of the program that will potentially be upgraded or modified in later versions. Describe provisions, if any, that have been made to facilitate those alterations, and give guidelines how such alterations can be installed into the program.

7.6 Error Message and Diagnostics

Provide a listing of all error messages, the conditions that invoke each, and the reason why such conditions are improper (if not obvious from the message).

7.7 Detailed Formats

7.7.1 Detailed Control/Response Message Formats

Provide detailed formats for each control message, together with the response(s) to that message as supplementary material to Section 4.4.1.

7.7.2 Detailed Data Input/Output Formats

Provide detailed formats for data inputs and outputs, together with the associated response(s) as supplementary material to Section 4.4.2.

7.7.3 Data-Base Input/Output Formats

Provide detailed formats for each external data base or file used to input data to or accept data from the program when appropriate. Identify each format with the corresponding data storage device listed in Section 3.2.7. Include descriptive data, such as: (1) number of characters per file record, (2) file data format, (3) number of file records, (4) special end-of-record marks, (5) special end-of-file marks, as appropriate.

7.7.3.1 Data-Base Input/Output Parameters

For the mass-storage devices listed in 3.2.7, define input/output parameters, such as: (1) structure and format; (2) syntax; (3) lengths and frequency; (4) device assignment by message; (5) special features such as access keys, passwords, or lockout keys; (6) special timing constraints; (7) special considerations such as online vs. offline operation; (8) data units and ranges.

7.7.3.2 Data Base Storage Device Data Control Characteristics

Define the characteristics of data control parameters, associated with the data, such as: (1) verification codes, (2) index tables, (3) list structures.

7.7.4 Communications Line Input/Output Formats

Provide detailed structure and format of all data blocks to be input from or output to data communication circuits, except for data lines to/from operator-control devices covered above. Reference applicable documents (7.2) where appropriate.

7.8 Auxiliary Tables

7.8.i Detailed Design Tables

Assemble in tabular form all auxiliary reference data needed for program specification, which is better located in an appendix rather than in the text proper. Display each of these as a separate subsection, 7.8.i.

7.9 Special Maintenance Procedures

Identify any special supporting software, documents, procedures, etc., used for maintaining the program, such as for debugging, testing, verification, QA, automatic redocumentation, etc. Give detailed procedures, guidelines, or hints for maintaining the program, as appropriate. Such procedures, if extensive, however, may well form a separate Maintenance Manual, in which case, only a citation to the proper reference in 7.2, above, need appear. See Appendix K for a detailed set of topics.

7.10 Decision Log

Enter and discuss each of the major design decisions that may affect program sustaining and maintenance.

7.11 Linkage Editor and Job Control Code

Describe the detailed linkage-edit code, job-control code, or map processing necessary to collect, load, and execute the program. If maintenance procedures require altering this code, instruct the reader how such changes are to be made.

8. CODE LISTINGS

The code listings form the final part of the "as-built" software specification. For large programs the listings will form a separate volume.

PLACING PAGE CLARK NOT HERE

APPENDIX F

USER INSTRUCTION MANUAL TOPICS

This appendix contains an outline of topics typically considered for inclusion in a software user's manual. The items listed are not exhaustive, nor are all of those given necessarily applicable to a particular given user guide. Rather, the topics herein contained are those that should be considered as candidates for inclusion in a user guide for a specific application. The demands and needs of users, as well as the type and cost of software capability being exposed, should dictate the level of detail, the arrangement of the material, the orientation of the presentation, and the scope of the content. The outline below is an attempt at providing a logically and hierarchically arranged checklist.

This text has repeatedly recommended that the user manual be written at least in a skeletal form from the top down (in detail hierarchy) concurrently with the writing of the SSD hierarchy and with the construction of the program, so as to provide timely information among developers, to permit the user manual to be tested concurrently with the program, and to avoid last-minute efforts to complete the documentation prior to software delivery. The emphasis in writing the user manual is on providing complete and effective information for exercising all of the options and capabilities of the program. The timely gathering of information and writing technical material for the manual, however, must not be put in series with the formal, more clerical aspects (such as typing and reproduction) of a documentation activity.

As the program construction proceeds in a top-down manner, usage information in greater and greater detail typically becomes available. If compiled and written into the user guide during this time, the information level will tend to aid in assessing whether the emerging program falls within its required capabilities implemented so far.

Figure F-1 is a top-level view of the suggested document organization; greater hierarchic detail is provided in the written outline that follows. This outline contains guidelines after each topical heading for the type of material to be inserted at that point. In full, the topics constitute Class A detail.

The users of the program assumed in this outline are not envisioned to be the operators of the program. The user determines the feasibility of the program to fulfill his needs, generates or prepares data (or causes it to be prepared), submits it for operations (either conversationally, interactively, or in batch), and uses (interprets) the output, if any, for an intended task. (In some cases, such as where a user causes a data base to be updated, that output may not be immediate, and may not even be a result of operating the program being described; in other cases, the task may be the gaining of insight into a problem.)

A topical outline for the operational manual appears in Appendix I. In cases where it is desirable to combine both user and operator functions into a single manual, the outlines can be merged appropriately.

Sources for the material contained herein are [44] and [45].

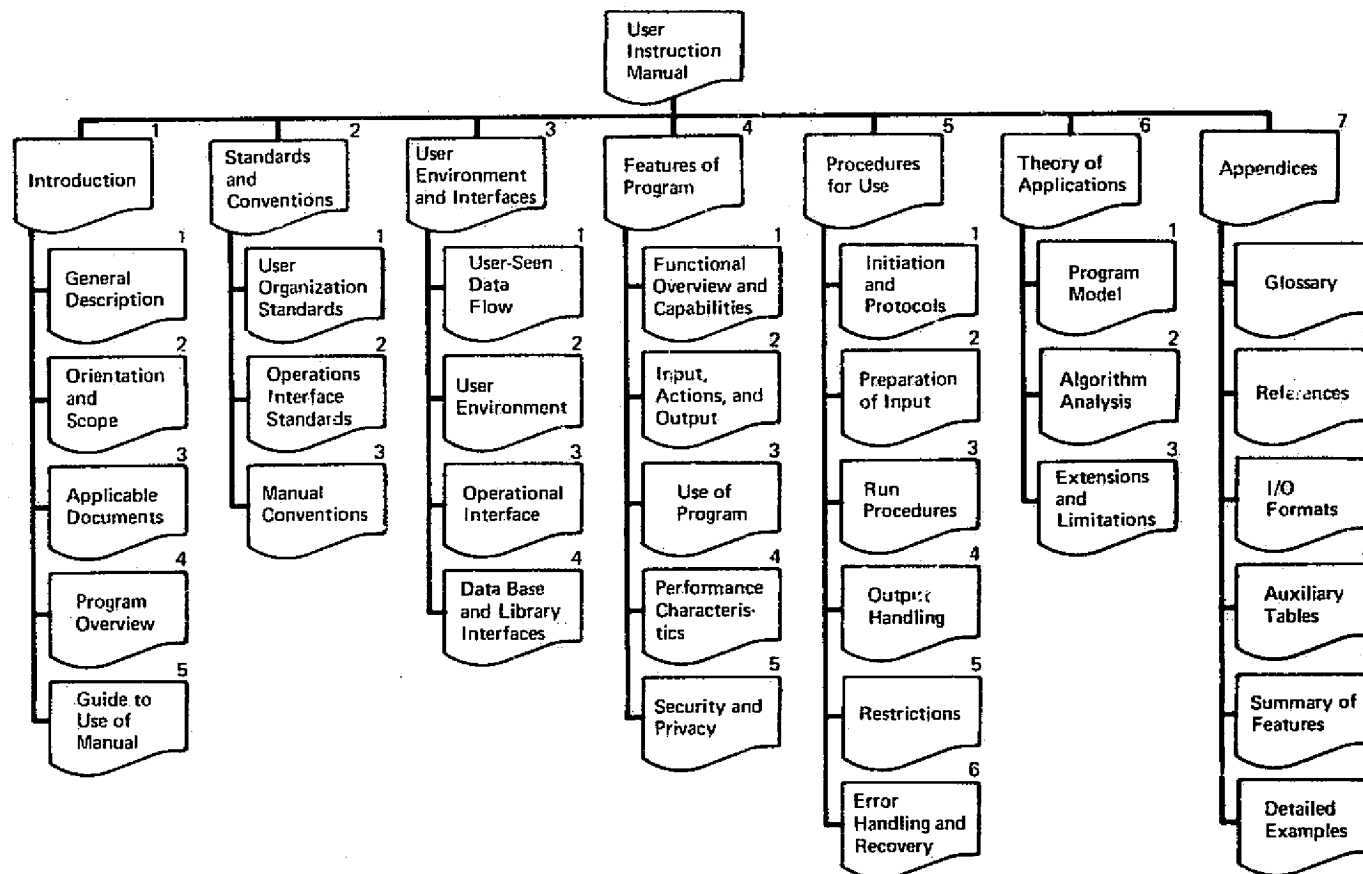


Figure F-1. Graphical outline of the User Instruction Manual

USER INSTRUCTION MANUAL

Detailed Table of Contents

FRONT MATTER

Title Page. Provide a title page containing (1) document number; (2) USER MANUAL; (3) program, subsystem, and system titles; (4) the publication date; (5) author and management authority signatures, as appropriate; and (6) releasing organization. Signature or publication approval should be supplied only at SSD completion. The date reflects the time of latest change to any item in the manual.

Abstract. Give a brief abstract that summarizes the purpose and usage of the manual.

Change Control Information. Provide a statement that specifies the current level of change control authority and describe procedures for submitting change requests and reporting anomalies.

Distribution Information. Provide information that tells how copies of this document may be obtained.

Table of Contents. Provide a detailed table of contents for the manual, which lists section number, title, and page of every item with a heading (this is probably the last-supplied item in generating the manual).

TEXT OF MANUAL

1. INTRODUCTION

1.1 General Description of the Program and Its Use

Provide a brief statement that describes the purpose and use of the program. Perhaps also appropriate in this introduction are background information, history, relationships to other programs or systems, and major applications areas.

1.2 Orientation and Scope

Identify the intended readers of this manual, their backgrounds, their assumed levels of data-processing expertise, and the extent to which the

content of this manual is self-contained. Describe the scope of the manual as it pertains to the usage of the program and its products. Identify any significant limitations of the program in applications.

1.3 Applicable Documents

Identify all documents, controlling or informational, that apply to or regulate the usage of the program.

1.4 Program Overview

Provide a brief functional description of the program and its intended usage, including the nature of the problem it solves, the philosophy and method of solution, and the type and content of data input, processed, generated, or transmitted. Identify the usage characteristics of the program, such as: (1) major applications; (2) real-time, interactive, or batch; (3) computational or data manipulation; (4) developmental or operational. Identify the general system, subsystem, and environment (hardware and software) in which the program operates (a block diagram is useful here) insofar as these considerations affect usage of the program; leave details to Section 3.

1.5 Guide to the Use of the Manual

Explain how this guide is to be used in applications.

2. STANDARDS AND CONVENTIONS

This section describes the standards imposed on or by the using organization, and the conventions (e.g., notations and terminology) applied in this manual.

2.1 User Organization Standards

Identify or reference applicable existing organizational usage standards, state any exceptions to these standards necessitated by the usage of this program, and provide any special standards required to use the program effectively.

2.2 Operations Interface Standards

Identify or reference existing standards that apply to the user/operations interface, state any exceptions to these standards (negotiated as required for operation of this program), and provide any special standards required for users to interface properly with operations.

2.3 Manual Conventions

Define notations, terms, and other conventions or assumptions used generally throughout the manual. Include such items as ways of

distinguishing literal fields from syntactic variables in descriptions of input and output formats, means for differentiating user inputs from outputs in examples of interactive operations, non-standard mathematical usage, special acronyms, etc.

3. USER ENVIRONMENT AND INTERFACES

3.1 Data Flow

Describe data sources and sinks, the flow of data from sources to sinks, and the role the program plays in this flow. Identify operations, systems, library and support interfaces, and describe their role as seen by the user. A diagram may be useful for illustrative purposes here.

3.2 User Environment

Introduce the general environment within which the user interfaces with the program. Identify interfaces among users (if any), location of users, their sources of data, the media through which they prepare data, submit runs, and receive output, their manual tasks, relationships among data in the user environment, etc. Describe user interfaces with management, if appropriate. Defer user procedures, specific formats, units, etc., until later sections.

3.3 Operational Interfaces

Identify and describe the interfaces between the program user and the program operational environment. Discuss, as appropriate: forms, input media, control media, interfacing procedures, data generation methods, storage media, modes of delivery of output to users, manual tasks, etc. Identify those items that are unique to this program and not covered by an overall system description or governing document (if this manual need not be self-contained). Defer operational procedures, formats, units, etc., to the operational manual (Appendix I), unless user and operational guides are combined in one manual.

3.4 Data Base and Library Interfaces

3.4.1 Data Base Interfaces

Describe all data files in the data base that are referenced, supported, or kept current by the program, insofar as these are visible to users of the program. Include the purpose of each such file, but defer detailed formats (if necessary for use) to an appropriate appendix. If there are offline or manually maintained parts of the data base that are pertinent to the usage of this program, similarly describe these elements.

3.4.2 Library Interfaces

Describe any appropriate user interactions or interfaces with document libraries, software (program, subprogram) libraries, or offline storage libraries other than those described in the operational interfaces above. Reference source documents for data preparation and editing aids, output data monitor and diagnostic aids, etc., as applicable.

4. FEATURES OF THE PROGRAM

This entire section documents the end-to-end functional characteristics and usage of the program, to a level of detail sufficient for stand-alone reference. This section should describe each functional capability and option of the program fully, giving examples of each, annotated and explained. Usage of graphic material in explanations is encouraged.

4.1 Functional Overview and Capabilities

Prior to detailed usage characteristics in the other subsections of this section, present an overview of program capability. Typical coverage at this point might address the structure, I/O and data flow, functional categories, range of applications, major operating modes, program configurations, utility factors, security/protection measures, etc., as seen from the user viewpoint. If graphics are used, support each with narrative explanations.

4.2 Input, Actions, and Output

Describe each feature of the program, as visible or of interest to the user, in sufficient detail that the user may apply the procedures for use contained in Section 5. Correspond features or combinations of features of the program to applications as necessary, and describe the options that may be exercised in each case. Cite and illustrate the advantages to be gained through use of these features in applications. (Input, processing, and output characteristics pertinent to usage will normally be integrated together in a narrative fashion, feature by feature. However, the outline below is segmented into three separate subsections so as to delineate specific items for discussion.)

4.2.1 Input Characteristics

Define the requirements of collecting and preparing user input data, parameters, and controls. Typical considerations are: (1) purpose or conditions—e.g., to make needed revisions to data base; (2) frequency—periodically, randomly, or as a function of an operational situation; (3) origin—network operations, program office, budget data base, spacecraft sensor, etc.; (4) accuracy required for meaningful output; (5) medium—punched card, manual keyboard, magnetic tape; (6) restrictions—amount of

data, priority, use authorization, security limitations; (7) quality control—instructions for checking reasonableness of input data, actions to be taken when data appears to be received in error, documentation of errors, etc.; (8) disposition—instructions for retention, release, or distribution of input data received.

4.2.1.1 Input Format

Provide the layout forms and syntax as necessary. Include a description of each entry, with adequate grammatical rules and conventions used in each case. Distinguish literal input from syntactic variable identifiers. Typical considerations include: (1) length, as characters/line or characters/item; (2) format, as for example, left-justified free-form with spaces between items; (3) labels, tags, or identifiers; (4) sequence, or the order of placement of items in the input; (5) punctuation, or use of spacing and symbols to denote start and end of input, of lines, of data groups, of items, etc.; (6) rules governing the use of groups of particular characters or combinations of parameters in an input; (7) the vocabulary of allowable combinations or codes that must be used to identify or compose input items; (8) units and conversion factors; (9) optional elements and repeated elements; (10) controls, such as headers or trailers.

4.2.1.2 Sample Inputs

Provide specimens of each type of complete input form. Such specimens should include, as applicable: (1) control or other header information denoting class or type, date and time origin, or function codes; (2) text or other input data to be processed by the program; (3) trailer, denoting the end of input and other control data; (4) indication of omissions, i.e., classes or types of data that may be omitted, or are optional; (5) indication of repeated data, i.e., classes or types of data that may be repeated, and the extent of such repetition.

4.2.2 Processing Characteristics

Describe the processing performed, including, as pertinent: (1) transformations, manipulations, and reductions on data; (2) accuracy or precision in computations; (3) sequences of actions; (4) logical concepts; (5) error checks and diagnostics; (6) provisions for recovery; (7) controls and options; (8) applications restrictions.

4.2.3 Output Characteristics

Describe each of the output forms or other program responses to the user in sufficient detail for his effective interpretation in stated applications. Typical considerations include: (1) use—by whom and for what purposes; (2) frequency—weekly, periodically, or on demand; (3) variations—modifications that may appear on the basic output; (4) destination—which

users or work area; (5) medium—printout, punched cards, CRT display; (6) quality control—instructions for identification, checks for reasonableness, authorization to edit or correct errors; (7) disposition—instructions for retention or release, distribution, transmission, priority, security handling, and privacy considerations.

4.2.3.1 Output Formats

Provide a layout of each user-pertinent output, with explanatory material keyed to the particular parts of the format illustrated. Include (1) header—title, identification, date, number of output parts, etc.; (2) body—information that appears in the body or text of the output, columnar headings in tabular displays, and record layouts in machine readable outputs, noting which items may be omitted or repeated; (3) units and conversion factors for numeric fields; (4) legends for abbreviated data; (5) accuracy; (6) trailer—summary totals, end-of-output labels, etc.

4.2.3.2 Sample Outputs

Provide illustrative examples of each type of output. In each case discuss (1) the meaning and use of control data applied; (2) the source and characteristics of the data processed; (3) pertinent facts about the calculations made by the software; (4) characteristics, such as the presence or absence of items under certain other conditions of the output generation, other ranges of input values, or different units of measure.

4.3 Use of the Program

Explain how the program and its features are to be applied over its spectrum of applications. Give selection criteria and specific, graphic examples that match the program inputs, actions, and outputs to their intended interpretations. Describe any human post-processing of the presented output which may be required for effective use of the program. State what inferences one may draw from output data, if any.

The intent of this information is to give the reader sufficient insight to judge whether the program applies to a particular problem; if it does, then this information should tell how to make the data into and out of the program correspond with the parameters and facts concerning his application.

4.4 Performance Characteristics

Describe the performance characteristics of interest to the user, including, where appropriate: (1) quantity of input and output; (2) throughput rate; (3) accuracy; (4) cost of service; (5) turn-around time; (6) reliability; (7) flexibility; (8) quality of service.

4.5 Security and Privacy

Describe security and privacy measures implemented in the program that restrict its usage or guard data integrity via authorization keys, priorities, protocols, etc. Instruct the user what features are operative within the several authorization levels, and identify penalties for inadvertent or malicious misuse. Provide warning and cautionary information, if applicable.

5. PROCEDURES FOR USE

This section describes how to prepare data or instructions to operations in order to apply the program to problems it can handle, or to achieve a desired processing and output. The material delineating input, run, output, and other procedures below can be integrated together to enhance readability, if appropriate, and, perhaps, integrated with the material of Section 4 as well, to match the feature descriptions with the procedures for their use.

5.1 Initiation and Protocols

Describe the user/operations protocols necessary to initiate a run, submit input, and receive output. Discuss, as appropriate: (1) opening a computer work order; (2) assignment and use of passwords and account codes; (3) authorization to use system and/or data base files; (4) assignment of permanent private files; (5) instructions for pickup or delivery of I/O material and running the program; (6) differences between interactive and batch protocols; etc.

5.2 Preparation of Input

Describe the procedures for gathering input data and putting it in the format required for running the program. Such procedures might include: (1) the method of extracting data from source documents or files; (2) usage of data preparation and editing aids or other software; (3) usage of special services, such as keypunch operators; (4) a checklist to determine rapidly if everything has been done; (5) special considerations for alternative input media; (6) special considerations for batch vs. interactive operation; etc.

5.3 Run Procedures

If processing requires or permits interaction or monitoring by the user, provide instructions for terminal operations. Describe (1) terminal setup or connect procedures, e.g., log-on; (2) loading and start up procedures; (3) data or parameter input procedures; (4) control instructions; (5) magnetic

tape operational procedures; (6) cassette tape device operation; (7) run interruption/recovery; (8) run abortion; (9) special terminal devices, e.g., plotters; (10) indications of anomalous behavior and corrective actions; (11) start, restart, and other precautions; etc.

5.4 Output Handling

Describe applicable policies and/or procedures within the user environment for handling, disposing, dissemination, and routing of the various forms of output; for storage or archiving of output items for their later retrieval; for status reporting based on output parameters; for extraction and summarizing of information; for audits or other inspections of the output data, deadlines, etc.

5.5 Restrictions on Use

Identify and explain exceptions and restrictions in the procedures for preparing input, using the program, or handling the output. Such material might address (1) limited availability of source data; (2) security considerations; (3) processing cost vs. time of day limitations; (4) restrictions on amount of input or output; (5) accuracy; etc. The restrictions and exceptions discussed here are restrictions in usage procedures, rather than in the program applications.

5.6 Error Handling and Recovery Procedures

If not adequately covered in other parts of the manual, describe (1) detection procedures for anomalous output data; (2) meanings of error messages, codes, or indicators; (3) prescribed corrective actions by the user; (4) procedures for correcting input errors, for restart/recovery, for reporting of anomalous behavior, etc.

6. THEORY OF APPLICATIONS

In some cases a short section revealing salient aspects of the program model is an advantage in being able to use the program. In such cases, describe in this section the theory required for effective use of the program.

6.1 The Program Model

If the program reflects a real-world situation implemented via a parameterized software model, describe the model, the assumptions necessary for model validity, the correspondence between program parameters and real-world values, the pertinent differences between the model and actuality, and comparisons with other models or programs.

6.2 Algorithm Analysis

If the actual programmed procedure or sequence of steps executed by the program influences effective application of the program, discuss the pertinent details of the algorithms involved. Such an analysis might address program/user interaction and sequence of operations for the various operating modes, accuracy of steps, non-convergence in unusual cases, circumstances under which the program runs inefficiently or requires large allocations of storage, circumstance under which processing is very efficient, etc.

6.3 Extensions and Limitations

If the program model or algorithms have sufficient generality for later extension to wider applications, or for modification to new application areas, and these qualities influence the way the program is used or chosen for an application, they can be described in this section. If there are limitations in the model or program algorithm that limit the applicability of the program, and which may, perhaps, cause the user to analyze the particular circumstances of his application as a result, then these restrictions can be similarly discussed.

7. APPENDICES

Appended material may include, but are not limited to, explanatory material and references of an auxiliary nature, inserted directly or bound separately for convenience. The following suggested topics are typical. Appendices may be designated as "Appendix A," etc., if desired, rather than by the Dewey-decimal system given here.

7.1 Glossary

Give a list of mnemonics, acronyms, and unfamiliar or specially used terms appearing in the manual; provide definitions for each.

7.2 References

Provide a bibliography of references to other documents appearing in this manual. For each, include a brief statement indicating the nature of the subject matter and purpose of the reference.

7.3 Input and Output Formats

Provide detailed formats and syntax for data inputs and outputs, together with associated response(s) as supplementary material to Section 4.2. Define, as appropriate: (1) data base I/O formats, parameters, and control

characteristics; (2) communications device I/O formats, parameters, and control characteristics; etc.

7.4 Auxiliary Tables

Assemble in tabular form auxiliary reference material needed for program usage that is better located in an appendix rather than in the text proper. Display each table in a separate subsection, 7.5.i, and introduce or explain the use of each table narratively.

7.5 Summary of Features

Provide an abbreviated description of each of the program features for the knowledgeable user. This summary should be devoid of tutorial explanations, containing, instead, only technical descriptions or definitive examples for quick reference.

7.6 Detailed Examples

Display the usage of the program via sample runs from beginning to end. Show all input, indicate all interactions in timely sequence, and display all output. Give examples of normal and abnormal runs, and illustrate the procedures followed in each case.

RECEIVED PAGE BLANK NOT FILED

APPENDIX G

CRISP SYNTAX AND STRUCTURES

Chapter 7 introduced a family of languages based on a common set of control structures called CRISP. The principal use of CRISP in this text has been to illustrate the isomorphism between CRISP structures and flowchart topologies, the use of a procedural design and description language that is not keyed to a specific coding language, and the natural conversion of these program descriptions into, perhaps, unstructured coding languages.

This appendix contains a functional description for three types of CRISP processors: CRISP-PDL, the design and documentation aid; CRISPFLOW, a graphic aid that flowcharts program algorithms; and CRISP translators, which overlay such unstructured coding languages as FORTRAN or assembly language and promote the one-to-one correspondence between design and code via commonality of syntax and form.

I have chosen to pattern this description in the form of a Software Functional Specification, using Appendix E as a guide for topical material. Front matter and such sections that are non-applicable have been omitted. To limit space, all of the Programming Specifications and Test Specifications normally found in an SSD have been eliminated from this appendix. Even the functional specifications that do appear admittedly lack concreteness and detail in certain particulars.

This appendix, as it stands, then, should be viewed as preliminary and embryonic in the development of a CRISP system. Nevertheless, the format, content, and level of the document are probably sufficient for the purpose of this text, both to define most of CRISP and to illustrate what an SSD might typically look like during the software development process. Other examples of SSDs appear in Appendix L.

CRISP SOFTWARE SPECIFICATION DOCUMENT

1. INTRODUCTION

1.1 Purpose of this Specification

CRISP is a set of Control Restrictive Instructions for Structured Programming. It consists of a few forms that conform program control flow into structured designs. Programmers using a CRISP system document their designs and write code using the most appropriate languages available, except for statements governing program control flow. CRISP statements are used in lieu of the control statements of the language being used.

CRISPFLOW is a form of CRISP that turns CRISP documentation into structured flowcharts. CRISP-PDL is a program design and documentation tool that allows all but CRISP syntax to be freely chosen (usually abbreviated English), and has as its output cosmetized indented listings, identifier cross-references, a tier chart, a glossary, a table of contents, stub status reports, and a statistical usage summary. Other CRISP processors governed by this specification translate input statements into an arbitrary given target language, with execution monitors inserted automatically when desired for correctness testing.

This specification sets forth the syntax and semantics of the CRISP forms; it is the controlling standard for all processors implemented in the CRISP family.

1.2 Scope of Applicability

This document specifies functionally that part of CRISP which covers the basic control structures (sequence, alternative selection, looping, and procedure linkages) and miscellaneous features such as macros, line continuation, paranormal exits, comments, and end of processing. This specification does not cover nor preclude possible subsets, supersets, or other options that may be implemented into a specific processor. However, all processors using the CRISP title must use features strictly in conformance with the minimum standards herein contained. The CRISP forms described here are implementation independent; neither syntax nor semantic value may be altered to benefit a specific processor design.

Section 2 of this specification contains standards and conventions used in this SSD. Section 3 discusses environmental and interface assumptions.

Section 4 forms the main body of functional specifications for CRISP. Sections 5 and 6 have been omitted as being implementation oriented, and Section 7 contains appended material relative to the functional behavior of CRISP only.

1.3 System/Subsystem Overview

Inasmuch as the features described in this document are meant to be machine and software-system independent, the programming and operational environment for CRISP are assumed to conform to a broad, generally non-restrictive set of standards. The assumptions that appear explicitly and implicitly herein have been made chiefly to foster the readability of this document rather than to set system requirements for an implementation. Environmental characteristics substantially differing from assumptions herein stated are permitted to govern so long as they do not lead to conflicts with syntactic and semantic specifications set forth in this document, nor produce viewable outputs with significant variations from those formats specified. Further details appear in Section 3.

1.4 General Description of CRISP

The quality of a computer program can often be significantly influenced by the design medium in which that program is developed, embryonically and evolutionarily, from the ideas that permeate the programmer's mind to the completed programming specification. The medium must foster the expression of such ideas easily and quickly (sometimes before they fade from memory), and must permit flexible and facile alterations, additions, and deletions to these ideas as the design evolves. Moreover, the expression of the design should be as graphic as a "picture of the program"—yet not be the program, nor constrained by the syntax of a computer language. At the final evolutionary stage, such descriptions should form the principal program design documentation.

A "program design language" or "procedural description language" is a formalized embodiment of such design aids, and can take many forms. Probably the most familiar form is flowcharting. This specification describes a procedure-oriented language type and processors for it, which result in program specifications and code that can appear much like flowcharts. Moreover, if proper attention has been paid to certain limitations, the CRISP statements can be flowcharted directly, using CRISPFLOW as described in Section 4.2.2.

A program design in CRISP-PDL consists of short, English textual descriptions of data manipulations, operations, and functions imbedded within structured CRISP control-logic syntax. The output listing of CRISP-PDL displays the program as modules of hierarchically refined algorithms

cosmetically formatted into 2-dimensional, flowchart-like segments. There is no restriction on the use of the English textual material, only on the usages of the program control logic, that must adhere to certain syntactic rules (to be further described later in this document).

CRISP translators access sequential source records and replace recognized control-logic statements by equivalent target language code that performs the specified control flow. Although the programming specification is not part of this document, it may be useful for the reader to envision this translation process in the form of macros invoked by the CRISP statements; each target language has its own set of "plug in" macro bodies specifically tailored to that language. Descriptions of the STAGE-2 general purpose macro processor (References 7.2.1 and 7.2.2) contain enlightening information regarding the use of such replaceable macro forms in portable programming applications. Alternatively, CRISP may form the control sublanguage of a fully compiled language.

CRISP preempts all control statements from a base language and substitutes forms that force programs to be structured in control. However, this specification does not require that CRISP-preprocessors analyze non-CRISP statements syntactically; hence, usage of non-CRISP control forms are not prohibited by this specification (but neither are they encouraged).

Programmers may thus construct code using statements from a background, or base language, such as, perhaps, FORTRAN or Assembly Language, except for statements that govern the flow of control (branching, looping, etc.). Such control is accomplished by the use of statements as specified in this document.

The source program thus consists of a mixture of CRISP control-structure statements and base-language code. A preprocessor for such a source program would then translate or otherwise process only those CRISP constructs back into the equivalent base-language-coded structures for compilation. Base-language statements would be passed to the compiler directly (possibly with some format adjustment).

All functions herein described make no assumptions relative to whether the CRISP processors are real-time, interactive, or batch operated. The processors are assumed, however, to be solely data manipulative, with reasonable restrictions on throughput rate and efficiency, and are not restricted to single-pass translation.

Translators may be preprocessors for a given base language, such as FORTRAN, BASIC, or Assembly Language, or may be full compilers for languages with CRISP as the control sublanguage.

2. STANDARDS AND CONVENTIONS

This section describes standards and conventions used in this SSD to describe the external characteristics of CRISP.

2.1 Specification Standards and Conventions

Flowcharts drawn by CRISPFLOW and illustrated in this SSD adhere to ANSI standards (Reference 7.2.3) as augmented or interpreted to fill the needs of structured programming, as put forth in Reference 7.2.4. As a further convention, binary decision boxes on flowcharts show the *true* flowline leaving on the left, *false* on the right. Multiple-decision symbols have the results displayed in case-order from the left. These conventions make top-down-left-to-right flowchart scanning correspond to top-down readability of the CRISP code structures.

Italicized lower-case identifiers in Section 4 of this document identify syntactic variables, for which a substitution must be made. Subscripts may appear on syntactic variables to distinguish multiple usages of the same type. Square brackets, [], enclose optional fields. The syntactic variable types are listed below:

comment: string of characters not containing "**>*"

date: string, presumed to be date

dewey: decimalized module identification number

event: interrupt label

eventlist: list of events

expression: target syntax numeric expression

fname: function name, target syntax

identifier: alphanumeric, possibly with periods and underscores

index: target syntax discrete-valued variable

integer: positive integer, no decimal point or sign

label: ABORT transfer point

moduleender: keyword ending a module

moduleheader: keyword and name beginning a module

name: procedure name, arbitrary string

pname: program name, perhaps in target syntax

predicate: boolean-valued expression in target syntax

statement: CRISP or base-language statement

string: arbitrary string of characters

structure: body of CRISP and base-language statements

subname: subroutine name, target syntax

template: macro name, string with escape identifiers

value: value of index or OUTCOME variables

xsubname: external subroutine name, target syntax

Procedural specification standards are to be supplied prior to commencement of the design activity.

2.2 Programming Standards

Programming standards are not addressed by this specification. To be supplied prior to commencement of coding.

2.3 Test and Verification Standards

Test and verification standards are not covered by this specification. To be supplied prior to commencement of coding.

2.4 Quality Assurance Standards

QA standards are not covered by this specification. To be supplied prior to commencement of coding.

3. ENVIRONMENT AND INTERFACES

3.1 System/Subsystem Description

As discussed earlier (Section 1.3), these CRISP specifications are very general with respect to the system environment. There are, thus, no applicable system/subsystem documents. Interfaces between the CRISP user and operations are also undefined by this specification. The user is, therefore, assumed to be responsible for the generation and preparation of source materials and for the following of standard procedures and protocols operative within the computation facility hosting the CRISP system.

Even though this CRISP specification is shielded from the operating system and hardware, certain assumptions about input, output, and storage media are necessary. The characteristics of these media and the constraints they impose on the program are discussed in Section 3.2, below. Interfaces with supporting programs are discussed in Section 3.4.

3.2 Hardware Characteristics and Constraints

The hardware functional characteristics, displayed graphically in Figure 3.2.1, include:

- A program core allocation
- A control-input medium (optional in this specification)
- A scratch-file capability
- A diagnostic output display medium
- An output listing device or display
- Target code output medium (for CRISP translators)
- Sufficient memory for tables and lists
- Plotting capability

Constraints imposed on the hardware by this specification are (1) that these media be available in a form judged to be suitable for the particular implementation; (2) that the amount of storage needed for such things as cross-referencing of identifiers, tier chart, table of contents, glossary, macro translation, etc., be sufficient that programs of significant size be accommodated. Information relative to these limitations may be determined separately for each implementation.

For the purposes of visualizing CRISP specifications, the reader may find it useful to envision these media assigned as follows: The source input comes from disk files named by the control-data input device, an interactive demand terminal, from which the other program options can also be input. The program scratch files are disk files, and the output medium is the system line printer. Diagnostics and monitor data appear at the user terminal. The CRISPFLOW plotter draws straight lines between points specified, and lines may be output in random order. The operating mode is either online or batch, demand or queued, and operates as a sequential program.

3.3 Software Environment

The software required for CRISP consists of an unspecified language compiler or compilers suitable to implement and maintain the source processing tasks described in Section 4 and to manipulate properly the media in Section 3.2, above. In addition, text editing and debug facilities are required for program developments using CRISP. No standards are hereby set by this document for these tools.

3.4 Interface Characteristics

Interface characteristics between CRISP processors and the system shall be defined by standards and protocols imposed by or negotiated with the operating environment. CRISP source media, however, shall be compatible as follows: All CRISP source data shall be accessible by CRISP-PI/L and

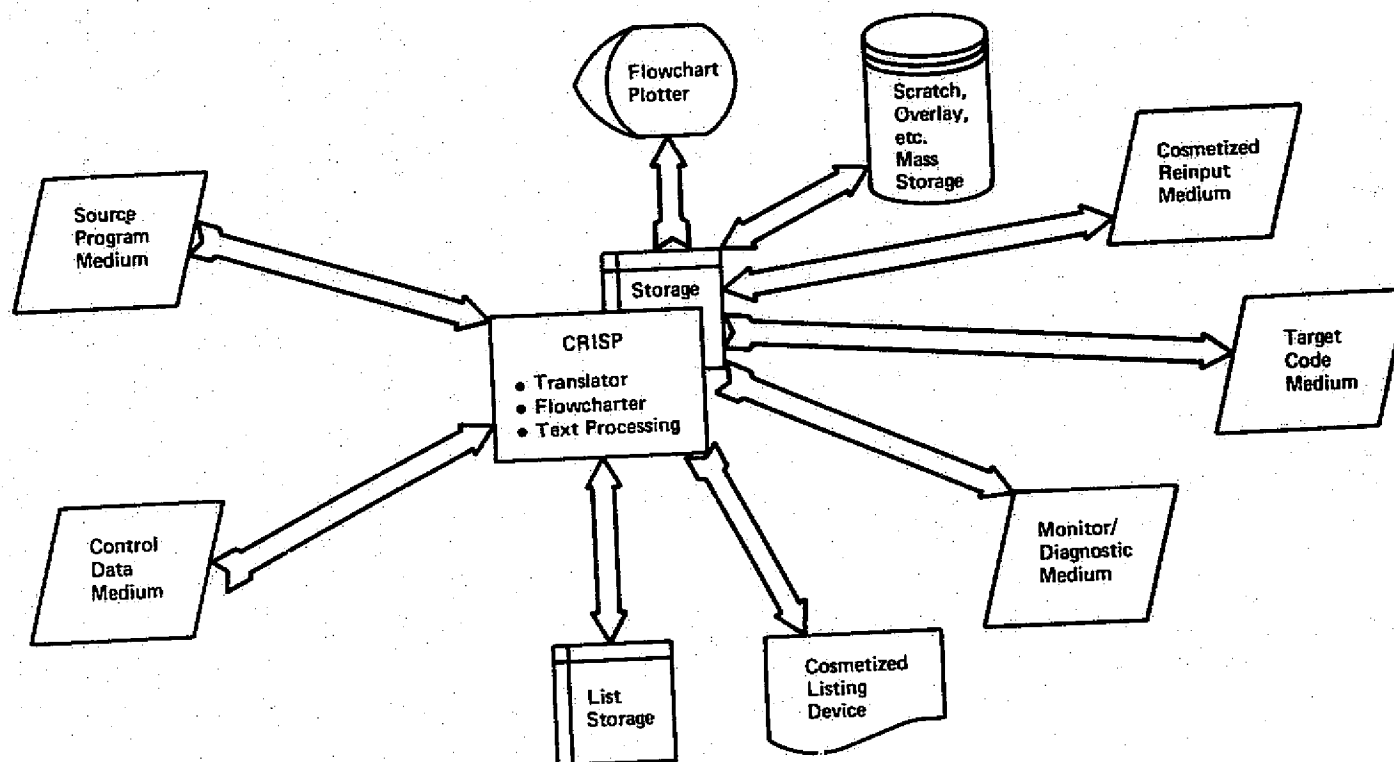


Figure 3.2.1. Hardware functional characteristics assumed for the CRISP system

CRISPFLOW for processing. That is, all source data can be processed by CRISP-PDL for indenting, cross-referencing, etc., and flowcharts can be generated from all forms of source input, as well (provided care is taken to assure that the amount of text and numbers of boxes do not yield flowcharts beyond readable limits).

4. CRISP FUNCTIONAL SPECIFICATIONS

4.1 Functional Organization and Overview

The discussion of the CRISP system will apply to CRISP-PDL, CRISPFLOW, and CRISP translators. The syntax of statements for each is the same, but the output varies according to the particular processor used. The CRISP editor is not covered by this specification.

Except for comments, CRISP constructions are keyword actuated. That is, the first symbol or symbols on each line uniquely identify whether a source statement is a CRISP form or belongs to the base language. Such keywords identify the beginnings of structures and substructures, linkages to other structures, or the end of structures or substructures.

Because the CRISP statements are keyword actuated, it is necessary that statements in the base language not begin with these keywords. CRISP structures can conceptually be iterated and nested to any desired level to produce the intended program. However, flowcharting or the use of indentation to identify levels of nesting and to promote readability tends to limit how many levels can actually be accommodated as a practical matter. This specification does not limit the maximum number of levels. The minimum number of levels that CRISP-PDL and CRISPFLOW can accommodate must be no less than 10.

In arriving at the structures and syntax herein contained, a number of concerns have been expressed and evaluated relative to alternatives within the purview of this specification. Some of the criteria that shaped this specification are listed below.

These criteria led to the selection of imperative-form verbs for naming actions to be taken, possibly modified by conjunctives (IF, UNLESS, WHILE, UNTIL). Alternative-selection structures are introduced by predicate-conjunctives (IF, in CASE). Almost all structures that introduce a level of nesting of statements within that structure conclude with an appropriate ENDxxx, where xxx identifies the type of structure being closed. (An error message will occur when a mismatch occurs between the structure in effect and the END-type.)

At one point during the generation of this specification, there was a concern whether CRISP statements should ignore blanks (spaces) in input—as does FORTRAN—or whether CRISP should promote code readability by requiring spaces in certain places as string delimiters. The final decision was that CRISP shall specify the use of spaces to delimit syntactic elements only in those places required by a preprocessor to discern keywords.

The list below is an ordered set of concerns that were considered in the formation of this specification. The order shown displays the approximate priority of that concern.

1. Minimum number of control structures consistent with programming efficiency.
2. Generality of application.
3. Clarity, readability, understandability of syntax.
4. Ease in assessment of program correctness.
5. Implications of automatic flowcharting.
6. Consistent form of syntactic elements.
7. Implementation ease (single vs. multiple-pass processing).
8. Ease of use, interactive and batch.
9. Blank-independence of syntax.

4.1.1 Overview of CRISP-PDL

The CRISP-PDL processor is principally a text-formatting, annotation, and cross-reference generating device (Figure 4.1.1.1). A procedure input line consists of possibly three fields: a prefix, a cosmetic, and the text. The prefix contains possibly a step number within the module (usually chosen to correspond to a box on a flowchart) and any subroutine or function flowchart cross-references. The cosmetic portion consists of spaces and vestigial flowlines, so as to present an indented listing, which then also displays many of the features of a flowchart.

In procedural descriptions, the text field of the input is of two varieties: either a control-logic text, or else a "base-language" (non-control-logic) text. The text field is distinguished as the field beginning with the first non-cosmetic character following the first space encountered on a line. Control-logic text fields begin with a keyword, such as IF, LOOP, REPEAT, or a left parenthesis "(".

Keywords signal the processor to increase or decrease the indenting level and to add, delete, or modify the vestigial flowlines. The headings of nested structures (e.g., IF, LOOP, ELSE, etc.) increase the indenting level and add

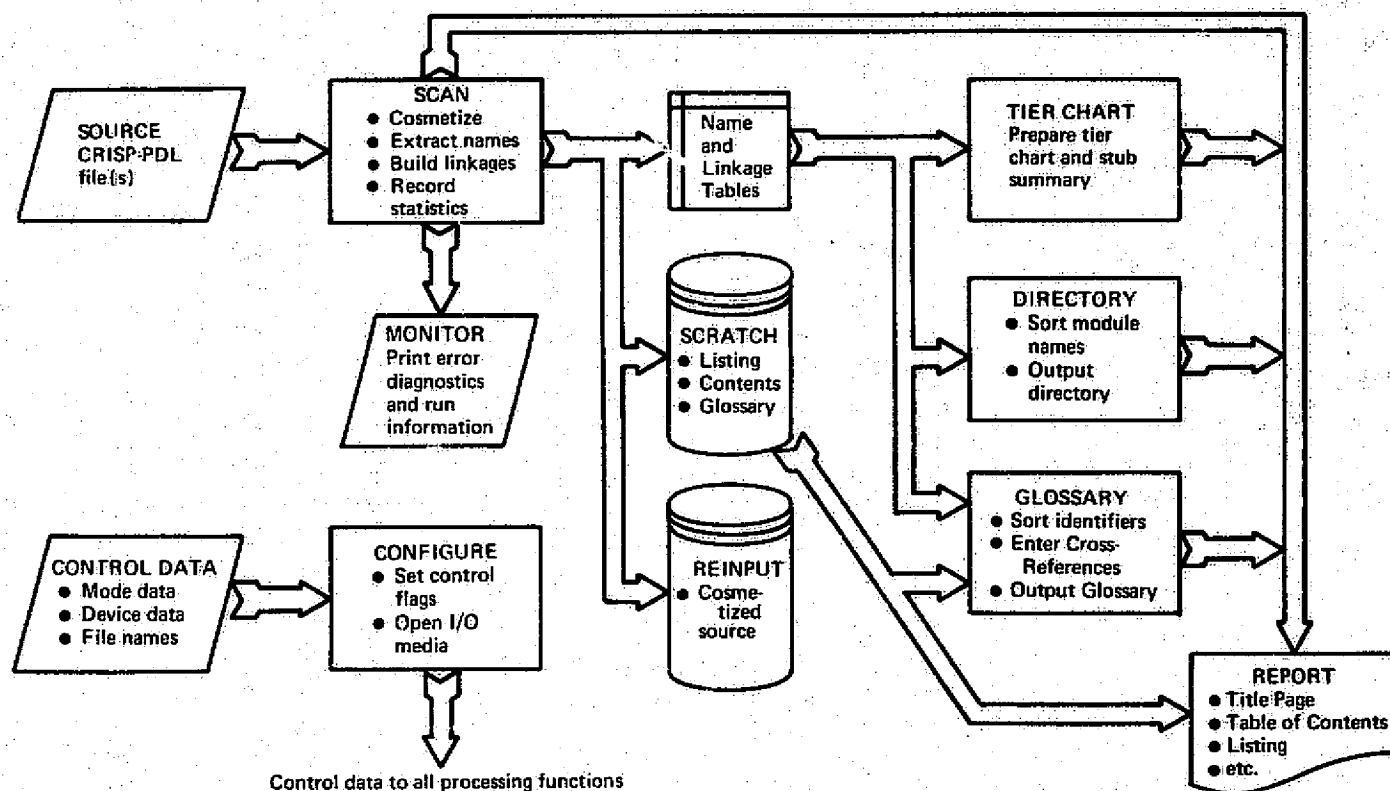


Figure 4.1.1.1. Functional data flow in the CRISP-PDL processor

flowlines; endings of nested structures (e.g., ENDIF, ENDCASES, REPEAT, etc.) decrease the indenting level and eradicate flowlines. Abnormal and paranormal exits (EXIT, RETURN, STOP, and SYSTEM) cause no change in indenting level, but do show a flowline exit of the current nesting level back to the appropriate level.

Since CRISP-PDL responds only to keywords, control structures of the base language will not produce extra levels of indentation or cosmetic flowlines.

CRISP-PDL also has features for cross-referencing procedure and data names, generation of a table of contents and tier chart, a completion status report, and keyword usage summary.

4.1.2 Overview of CRISPFLOW

CRISPFLOW accepts CRISP-PDL and other CRISP processor inputs for processing as control-logic structures and as arbitrary text strings, from which it then generates flowcharts (Figure 4.1.2.1). The CRISP logic structures determine the flowchart topology, and the text strings label the chart appropriately. Comments (Section 4.3.3, below) are not ignored in some statements (e.g., DO, CALL, CALLX) but are transferred into flowchart boxes or other chart annotations directly. Some comment fields, however, are discarded, as detailed in Section 4.2.2, below.

Since CRISPFLOW responds only to keywords in plotting control flow, use of base language control statements will not be detected, and, therefore, will not be flowcharted properly.

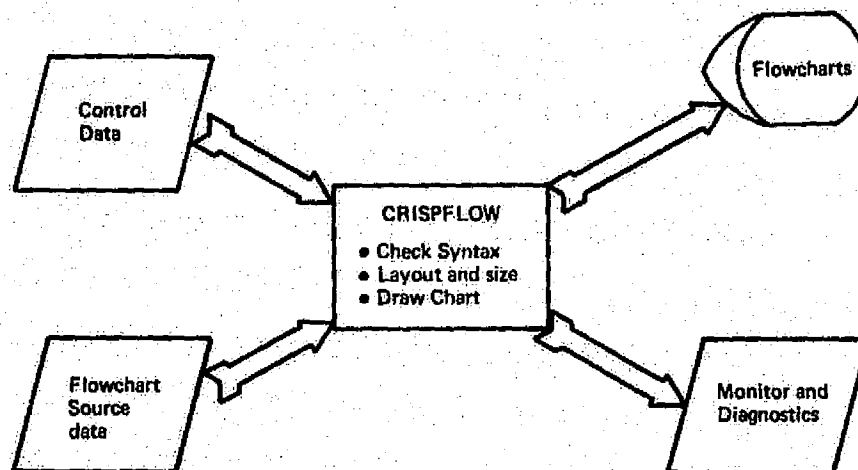


Figure 4.1.2.1. CRISPFLOW processor

Flowcharts are drawn to ANSI standards and as otherwise stated in Section 2.1.

4.1.3 Overview of CRISP Translators

The philosophy for CRISP translators (Figure 4.1.3.1) presumes that all comments, as detailed in Section 4.3.3, below, will first be removed from the source code, that the format of all CRISP constructs will then be checked for syntactic correctness, and that, as a minimum requirement, base-language statements and strings within CRISP statements will be transferred into direct target output without further syntactic analysis and without translation. Compilers based on CRISP, however, will perform full syntax checking and translation for all statements.

4.1.4 Detection of and Recovery From System Failure

System failure detection and recovery measures during CRISP operations are not covered by this specification.

4.1.5 Detection of and Response to Source Input Errors

As a minimum requirement, all CRISP translators shall detect errors of the following types: (1) missing elements in CRISP statements, (2) improper

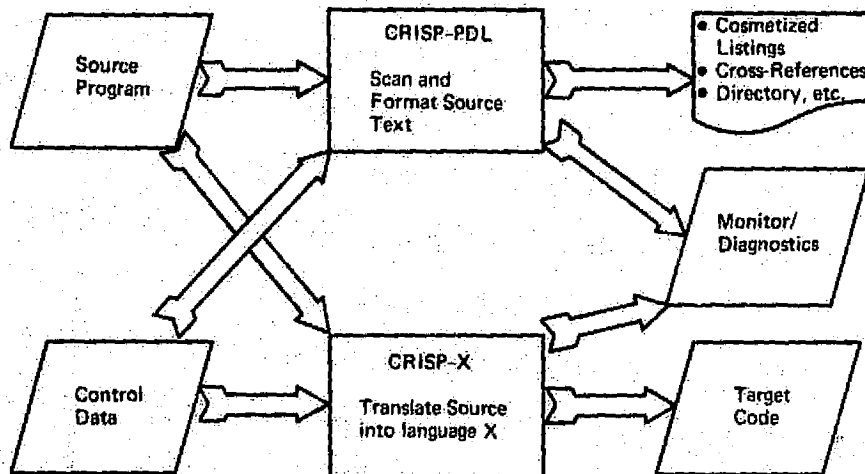


Figure 4.1.3.1. The CRISP translator generates target code, while CRISP-PDL is used for listings (see also Figure 12-2 of Reference 7.2.5)

elements in CRISP statements after a valid keyword, and (3) improper nesting of CRISP structures within a procedure unit. Specifically, no check is mandatory, as a requirement of this specification, to ascertain whether base-language control-logic mechanisms have been employed, rather than those specified here. CRISP-PDL detects only errors of type (3), above, and CRISPFLOW detects some errors of types (1) and (2) as necessary to determine flowchart structure.

Upon encountering such errors as alluded to above, the processor shall emit an appropriate diagnostic message and proceed. These error messages are to appear on the diagnostic medium preceded by the offending statement. The processor shall not stop upon error detection, even though a portion of its output may be in error.

In the case of nesting errors, the processor shall attempt partial recovery, for example, by scanning the nesting levels for the same type causing the mismatch, and assuming appropriate statements as `ENDxx:` to bring the processor back into some semblance of level-alignment. Statements or parts of clauses following `ABORT`, `CYCLE`, `EXIT`, `LEAVE`, `RETURN`, `SYSTEM`, `STOP` will be recognized as being unreachable, an error.

The exact type and degree of nesting-error recovery is not covered by this specification. The appearance of output listings may, therefore, show variances from implementation to implementation until standards for recovery are provided.

Diagnostic messages shall describe errors in the user input, and not the internal program methods used to discover the error.

4.2 Detailed Software Configurations and Modes of Operation

Whether the CRISP system is implemented as a single program with PDL, FLOW, and translator modes selectable by control data or whether each processor is a separate program is not specified by this document, but may vary among implementations. Furthermore, the specifications contained herein do not require any prescribed set of options among the processors.

All processors may selectively disable or abbreviate certain types of output according to the wishes of a user, input as control data. However,

there shall be no option that can alter the syntactic or semantic value of the forms given herein.

Output options are not distinguished as separate modes of operation in the coming descriptions, but merely as variants of the three basic modes.

4.2.1 CRISP-PDL Mode

Figure 4.1.1.1 is a second-level-of-detail HIPO diagram that summarizes the CRISP-PDL inputs, processing functions, and outputs. Inputs are the source file(s) being processed into a formatted listing and control data that selects which of the outputs is to appear. Processing consists of cosmetizing source lines, accumulation of module names and cross-reference material, and output of such material as directed by control data. Output consists of (1) a title page; (2) a table of contents (page order); (3) program directory (alphabetic order); (4) tier chart; (5) a listing of all stub-status modules; (6) an indented listing; (7) a glossary with procedure and identifier cross-references; and (8) a statistical summary of the program structure.

4.2.1.1 Identifier Cross-Referencing

In order for cataloguing identifiers, other than procedure and subroutine names, to occur, those identifiers must either appear first in a LET...BE form, or else contain interspersed periods or underscores amid alphanumeric characters (the first of which is a letter), as, for example, "A.B.3". Every occurrence of such an identifier will be catalogued according to page and line number, whether in procedure descriptions or intervening text.

4.2.1.2 Page Ejection

The appearance of #*n* beginning a line causes the ejection of a page if the remainder of the page contains fewer than *n* lines, when *n* > 0; the occurrence of #0 ejects to the next page unconditionally.

4.2.1.3 Intermodule Text

Text appearing between procedure modules (see Section 4.3.6: initiated by the appearance of a ## line, terminated by ###) is copied directly to the listing scratch file without cosmetization or diagnostic checking. Each line is scanned, however, for identifiers being cross-referenced.

324 Appendix C

If a line appears between modules and begins with "*", then the text of that line is added to the table of contents.

4.2.1.4 Boxed Comments

Within a module (see Section 4.3.5), a line containing only the characters "<*" (except for cosmetics) causes the lines following it to be turned into "boxed" comments, up to the occurrence of a line with only "*>" on it.

The input

```
<*  
This text forms  
a segment of  
comments.  
*>
```

would appear on output as

```
<*****>  
<* This text forms *>  
<* a segment of      *>  
<* comments.         *>  
<*****>
```

In converting text for "boxing," the first line and last lines are converted to <*****>, the width being appropriate to the boxed comments. Boxed comments are indented to the current nested level. Since the entire block of text must be processed to determine the box width, buffer size may limit the number of lines that may be so boxed. In such cases, each buffer load is boxed according to the maximum width of text in that buffer.

4.2.1.5 Update Provisions

In an "unstar" option, the asterisk in column 1 within a module (see Section 4.3.1) will be converted to a period. When this occurs, a comment is generated containing the current date, and affixed to the module ender statement (replacing any previous comment). In this way, the date of last approved update is recorded. The module header comment often contains the original design date.

4.2.1.6 Right-Justified Text

The appearance of a backslash on a line causes both the backslash and the text to its right to be right-justified. This is a convenience in recording revision numbers to statements, or indices for ease in locating certain features. If the backslash appears inside a block of statements being

scanned for "boxing," initiated by lines containing "<*" and ">," respectively, then only the text left of the backslash is used to determine the appropriate box width. The ">" is omitted on such right-justified lines.

4.2.1.7 Output Listing

Source lines input between a module header and module ender are transferred to a listing scratch file indented and cosmetized according to the type of structure in which nesting occurs. These are illustrated in Section 4.3. The statement-continuation signal (&) at the end of a line is not printed on the output listing, but is placed on the cosmetized reinput file, in case that file is saved for use as a future source input; continued lines are indented so as to distinguish them from the first line in the continued statement.

Each line of a module is counted and used to aid the generation of page numbers for the directory and table of contents. Each module will begin a new page, as will any narrative that follows a module end and a new module header. (Each module end signals a page advance.) The output report starts the CRISP-PDL listing at page 1 (the table of contents and front matter are given Roman numeral page numbers). Record numbers of the source file(s) are affixed to each corresponding listing line, at the left. The format of the listing is shown in 7.7.2.6.

4.2.1.8 Program Tier Chart

The program tier chart is a listing of the program structure and hierarchy. Within each module, all modules invoked by that module are listed indented to show subordination. The tier chart is prepared from data saved during statement scan processing and written onto a report scratch file, in the format given in 7.7.2.4.

4.2.1.9 Stub Status Report

After the tier chart is written, the saved cross-reference data is queried for missing procedures and subroutines, for modules identified currently as stubs, and for modules given for which no invocation has appeared. A listing of these is also written onto the report file, annotated by that current status. The format is shown in 7.7.2.5.

Each page of the tier chart and stub report is given the next consecutive Roman numeral. The tier chart follows the table of contents/directory in the output report listing.

4.2.1.10 Table of Contents and Directory

The only significant difference between a table of contents and a program directory is the key on which the module identifiers are sorted. When

sorted by page number, the report listing is the table of contents; when sorted alphabetically by module name, the directory results. When both are generated, the table of contents precedes the directory. The formats for these appear in 7.7.2.2 and 7.7.2.3. Both of these will appear before the tier chart. Pagination of this material will be Roman numerals.

4.2.1.11 Module and Identifier Cross-References and Glossary

Once the entire source program has been scanned, then all of the cross-references have been accumulated. At this point, the listing(s), glossary, and cross-references are generated. The formats for the glossary and cross-references are shown in 7.7.2.7.

4.2.1.12 CRISP-PDL Report Output

The output report, in its fullest form, consists of the following sections or parts, in order:

- Title page
- Table of contents
- Program directory
- Tier chart
- Stub status report
- Intermodule text
- Cosmetized source listing, first module
- Intermodule text
- Cosmetized source listing, second module
- ...
- ...
- Cross-references and glossary
- Usage statistics summary

The first four items listed above are given Roman numeral page numbers, and the rest, Arabic numbers. The formats for each of these sections or parts may be found in 7.7.2 of this specification.

4.2.2 CRISPFLOW Mode

The main thrust of the processing specification is already stated in the preceding sections of this document. Simply said, CRISPFLOW produces ANSI standard flowcharts. The flowcharts are drawn one to a page, with boxes and textual material sized to fit on such a page. CRISPFLOW produces structured flowcharts only, and does not recognize any control-logic directives except those prescribed in Section 4.3.

Flowlines connecting boxes conform to standards set forth in Section 2.1, in that decision collecting nodes are located directly under the vertex of their corresponding predicate node, and all branches are labeled in case order from the left. Loop-collecting nodes and decision-collecting nodes are distinguishable (the latter being filled in, the former not). Loop-exit flowlines lead to subsequent procedures in vertical alignment with loop entry.

There are no off-page connectors drawn by this version of CRISPFLOW; all chart symbols are thus scaled to fit on a single page, reducing the size of chart symbols accordingly. No provision is made by this specification for not charting unsuitably sized modules, although a notification may appear on the diagnostic display medium in such cases. Implementations may elect to draw off-page connectors for CASES without violating this specification.

The flowchart symbols ("boxes") plotted by CRISPFLOW will have variable dimensions, but the standard width:height aspect ratios required by ANSI are satisfied. This allows for the potentially different lengths of text of each statement to fit into its box most appropriately. The letters "T" and "F" are drawn to annotate the *true* and *false* branches from every binary decision box (which go to the left and right, respectively). The index of a CASE clause appears to the left and below the intersection of the clause's input flowline with the "distribution bus" flowline.

Comments are discarded by CRISPFLOW, except those following a DO, CALL, or CALLX invocation, following IF or CASE predicates, in AT and FORK structures, or appearing in a module header or ender. In invocation statements, the text of the comment is inserted into the corresponding striped box as additional annotation. The date is reproduced at the right top of the flowchart page to aid in location of charts; this date is the text string found in the comment field of the module header or ender (both are entered, if present). No analysis is made of this string to verify its correct format as a date. Comments after IF and CASE predicates appear as annotations to flowlines.

Identification numbers and cross-reference information appear above the right and left edges of their box, respectively, as regulated by ANSI standard conventions.

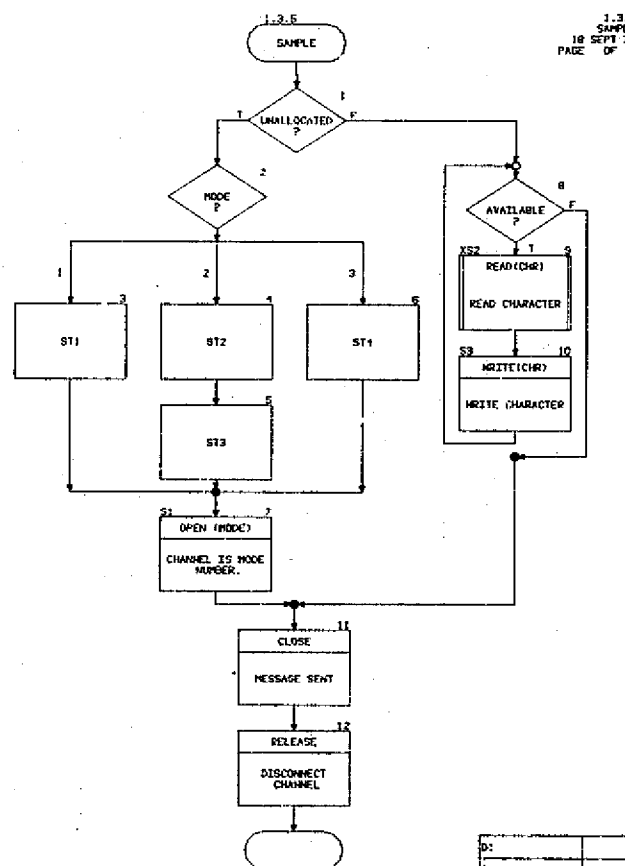
Text from all source lines continued using an ampersand (&) as the final non-blank character is inserted into a single flowchart box on output (cosmetics removed). Output will be separated into multiple lines within the box, broken at appropriate spaces. The ampersand does not appear on the output. The SAMPLE program charted in Figure 4.2.2.1 illustrates most of these features. Other outputs of CRISPFLOW are shown in Section 4.3.

PROCEDURE: SAMPLE <*18 SEPT 75*> MOD# 1.3.5

```

.1  IF (UNALLOCATED)
.2    CASE (MODE)
.3      (1)
.4        ST1
.5      (2)
.6        ST2
.7      (3)
.8        ST3
.9      (4)
.10       ST4
.11     ENDCASES
.12     CALL OPEN (MODE) <*CHANNEL IS MODE NUMBER.*>
.13   (ELSE)
.14     LOOP WHILE (AVAILABLE)
.15       CALLX READ(CHR) <*READ CHARACTER*>
.16       CALLX WRITE(CHR) <*WRITE CHARACTER*>
.17     REPEAT
.18   ENDIF
.19   DO CLOSE<*MESSAGE SENT*>
.20   DO RELEASE<*DISCONNECT CHANNEL*>
.21 ENDPROCEDURE

```



DS	
CS	
AS	

17 JUN 78

Figure 4.2.2.1. SAMPLE procedure illustrating flowcharting features

4.2.3 CRISP Translation Mode

The CRISP translator accepts a source input stream, recognizes and extracts elements of keyword-initiated statements and structures, and outputs target-language code in replacement. Translators need not be single-pass processors; in fact, `REQUIRE` and `MACRO` features require more than one pass.

Base-language source statements are first stripped of comments and then passed directly into target output. Some implementations may elect to attach the comment in appropriate target syntax to the output statement. If a base-language statement has been continued over many lines using the CRISP convention, and if the target language permits line continuation, then appropriate translation will be made. However, the user is considered responsible for otherwise ensuring that base language elements will be correctly interpreted by the target computer.

In some implementations (e.g., CRISP-assembly-language processors), some elements within CRISP statements not recognized as part of this specification may be parsed, so as to make effective use of the CRISP formatting. Notable in this class are predicates in `IF` and `LOOP` structures, and argument-passing devices for subroutines and functions.

However, in general, this specification only extends to implementations where features can be reasonably accommodated. Implementation of `FORK...JOIN`, for example, may not be implementable on systems not adaptable to concurrent processing.

Features in this specification may be, therefore, deleted to form CRISP subsets. However, no translators may introduce alternate or additional control forms and remain consistent with this specification.

Semantics and syntax of each CRISP form are contained in the detailed language specifications in Section 4.3, below.

4.3 Detailed CRISP Language Specifications

This section specifies the CRISP structures, their syntax, and the semantic value of each. Each construct is illustrated with a listing-source form (which may also be used as reinput to CRISP processors) and a corresponding flowchart. The listing forms show an indented format that also contains vestigial control-flow lines to aid in readability. The notation shown was dictated by the limitation of listing symbols to the ASCII set of characters. Listing media with other character sets may well have alternate cosmetics. The listing form shown is otherwise a part of this specification.

The structures specified in this document need not all be implemented in any given processor at any one time. However, any processor implementing a particular control structure herein covered must adhere to the details of this specification for that structure. CRISP-PDL and CRISPFLOW processors must, in addition, accommodate the superset of all CRISP subsets implemented.

4.3.1 The CRISP Statement

A *statement* is normally a single physical source line or input record. However, a statement may continue over more lines, if needed. Statement continuation is signalled by the appearance of an ampersand (&) at the end of a line, which indicates the next line is a part of the current statement.

Source lines may or may not be indented or have cosmetic annotations in them; these are ignored by processor. In addition, source statements may have a statement number and cross-reference designator specified, in the format

.statementnumber/crossreference cosmetics statementproper

Statement numbers are usually integers, and the cross-reference is usually an integer or alphanumeric; no spaces are permitted with this format. Either or both of these fields may be present or absent; if the statement number is absent, the period may also be absent; if a statement number is present, the period will appear in column 1; and if only the cross-reference field is present, the virgule (/) will appear in column 1. An asterisk may alternately be used for the period in column 1, when "change bars" are used to indicate differences between current and previous versions of a procedure. CRISP-PDL and translators do not process *statementnumber* or *cosmetics* fields, but recognize them so as to be compatible with CRISPFLOW. Additionally, translators use the statement and module numbers in REQUIRE and DISPLAY. See Section 4.2.2 for details on usage of these fields by CRISPFLOW and Sections 4.3.11 and 4.3.26 for usage by DISPLAY and REQUIRE.

The *statementproper* of an input statement begins either in column 1 or with the first non-cosmetic character after the first space (or other cosmetic on a line). Comments are not considered cosmetic. The characters that are considered in this specification to be cosmetic are: space, ",", "!", "_", "-", "<" not followed by "*", ">" when not preceded by "*", and "." when not in column 1. Implementations not having these characters may make suitable substitutions. Characters "!", "-", "<", and "." are not considered cosmetic in continued lines.

4.3.2 CRISP Keywords

The set of CRISP keywords (capitalized in the remainder of Section 4.3 for emphasis) are:

ABORT	ENDCASES	FUNCTION	REPEAT
AT	ENDFUNCTION	IF	REQUIRE
CALL	ENDIF	JOIN	RETURN
CALLX	ENDMACRO	LEAVE	STOP
CANCEL	ENDPROCEDURE	LET	STUB
CASE	ENDPROGRAM	LOOP	SUBROUTINE
CYCLE	ENDSUBROUTINE	MACRO	SYSTEM
DO	ENDTO	NORMAL	TO
DISPLAY	ENDWHEN	OUTCOME	WHEN
ELSE	EXIT	PROCEDURE	Left parenthesis, (
ENABLE	FINISH	PROGRAM	Percent, %
ENDAT	FORK		

These specifications present keywords as capitals; however, on implementations that permit lower case and/or underscoring, keywords may appear on the output either in upper or lower case and, perhaps, underscored or otherwise emphasized, depending on the implementation. When available, upper and lower case characters have equal syntactic value.

All keywords are terminated by space, left parenthesis, colon, or an end-of-line. The statement keyword must appear as the first non-cosmetic entry of a statement. Keywords may not be abbreviated.

4.3.3 Comment Delimiters

Comments in CRISP are denoted by the delimiters "<*" and "*>". Comments may appear anywhere within a statement after the keyword, if any.

The final "*>" need not appear if the comment extends to the end of a statement. Comments may extend over any number of lines when the line-continuation signal appears (see Section 4.3.3, below). The opening delimiter "<*" may appear inside a comment.

An arbitrary number of input lines preceded by a line containing only "<*" followed by a line containing only "*>" will be recognized as a comment block, and need not use line-continuation signals (&). See Section 4.2.1.4 for CRISP-PDL formatting of comment blocks.

332 Appendix G

Comments are scanned by CRISP-PDL for identifier references, and some comment fields are used by CRISPFLOW. See Sections 4.2.1.1 and 4.2.2, above, for details.

4.3.4 Non-CRISP Statements

Source input statements that do not begin with one of the above-named keywords are permissible, and are processed directly into the output (with cosmetics and indentation added in CRISP-PDL).

4.3.5 Program Modules

Several keywords (PROGRAM, PROCEDURE, TO, FUNCTION, MACRO, and SUBROUTINE) initiate program segments, hereinafter called "modules," which extend to their corresponding END statement. The initiating statements are herein called "module headers" and the closing statements, "module enders."

A typical example of the header syntax is the following:

```
PROCEDURE: name    [<*date1*>]                                [MOD#dewey]
```

Module-header statements may contain a cross-reference number following the optional substring "MOD#", continuing to the end of the statement. These are not used by the CRISP-PDL processor but by CRISPFLOW (Section 4.2.2) and translators (Section 4.2.3). A typical module ender syntax is the following:

```
ENDPROCEDURE [<*date2*>]
```

The header and ender *date* fields, being comments, are not used by CRISP translators but by CRISPFLOW to label flowcharts and by CRISP-PDL to record design and update times. The module flowchart format is defined in Figure 4.2.2.1.

Modules may not be nested within other modules, but may be linked to via DO, CALL, CALLX, and function invocations discussed below.

The general format of a module listed by CRISP-PDL is

```
moduleheader
    structure
    structure
    .
    .
    .
moduleender
```

The listing indentation shown is twelve spaces, to account for a *.dda/aaaas* field at the left margin, where *d* represents a digit, *a* represents an alphanumeric, and *s* denotes a space. The *module header* begins flush with the left margin. On abnormal and paranormal exits from a module, column 12 will contain a colon (:), as in Section 4.3.7, below. The module ender begins in column 13 except when abnormal or paranormal exits are present, in which case it begins in column 12.

A *structure* is a body of CRISP consisting of one or more statements or modular forms as detailed in the succeeding subsections of this section. Output listing cosmetics for structures nested within structures retain the cosmetics of the outer structures.

4.3.6 Intermodule Text

Textual material in the source input prior to the first module header, or between a module ender and the next header, is permitted, provided such text is preceded and followed by lines containing the characters *##* and *###*; such lines are treated the same as comments, in that CRISP-PDL examines this text for occurrences of *LET...BE* statements and identifiers containing period and/or underscore as spacers, and for lines beginning with *#* followed by text. Both CRISPFLOW and CRISP translators ignore such text. See Section 4.2.1 for further information.

4.3.7 ABORT Statement

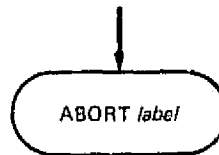
The ABORT statement may appear within a module:

```
moduleheader
...
...
:-----ABORT label
:
:
moduleender
```

In the listing, the proper number of *"-"* to fill the indentation level at the abort nesting level will appear past column 15. All subsequent lines will have *"?"* to denote an exit flowline in column 12. The *module ender* begins in column 12 when an ABORT has appeared.

334 Appendix C

CRISFLOW draws the terminal symbol with *label* printed inside, as below:



CRISP translators generate program transfers to the *labeled* statement elsewhere in the code, with proper subroutine/procedure linkage-recovery as may be necessary. The *label* is not checked syntactically as part of this specification.

4.3.8 AT Structure

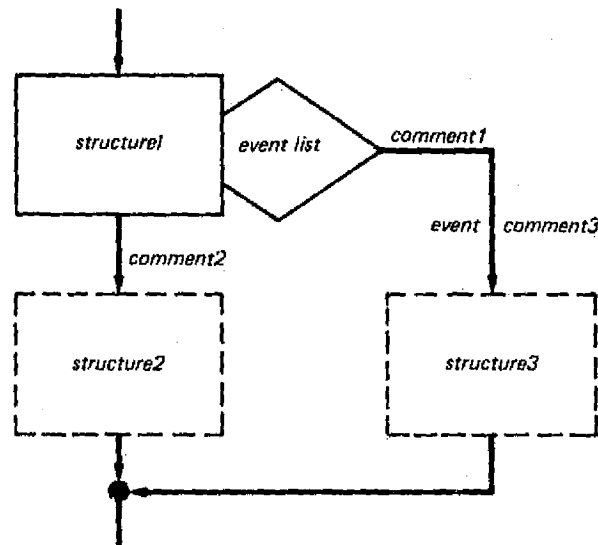
The AT structure sets up linkages to procedural blocks that may be initiated by contingency events occurring within the first body of code.

```
AT (eventlist) [<*comment1*>]
:   structure1
: -> (NORMAL) [<*comment2*>]
:   structure2
: -> (event) [<*comment3*>]
:   structure3
:   ...
: ..ENDAT
```

Any number of *structures* may appear, introduced by *events*. Each *event* appears in the *eventlist*, separated by commas if more than one *event*; *events* may be restricted in some implementations to appear in a particular order.

In order for the AT structure to be nestable within other AT (and, perhaps, WHEN) structures, it may be necessary for AT and ENDAT to save and restore interrupt vectors at runtime, or such other mechanisms used in the target language to assure program correctness in control.

CRISPFLOW produces the flowchart below for the AT structure:



The interruptible *structure1* is enclosed within a box as shown, drawn to proper scale (Reference 7.2.3). Substructures are drawn inside this box in a normal manner, except when that structure is a single statement. In this case, only the single box is drawn.

CRISP translators convert the *eventlist* into dedicated contingency interrupt arming target code (or equivalent) for such conditions as endfile, conversion error, overflow, etc. Execution of the NORMAL branch disables these interrupts, as does entry into any one of the contingency event code bodies. See Section 4.3.22 for further discussion of NORMAL.

4.3.9 CALL, CALLX, and DO Statements

CALL, CALLX, and DO statements invoke local subroutines, external subroutines, and singly occurring procedure bodies. The listing of such statements causes no indentation.

```

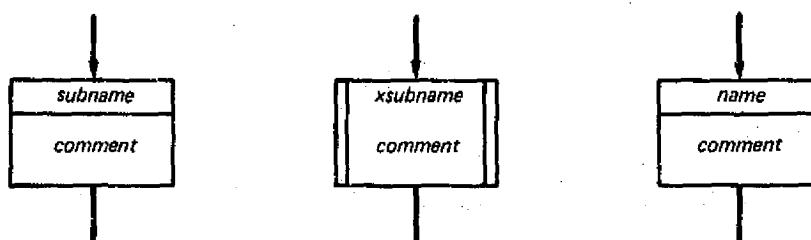
CALL subname [<*comment*>]
CALLX xsubname [<*comment*>]
DO name [<*comment*>]
  
```

The *subname* field must be in base language syntax for local subroutines (ones within this compilation); *xsubname* must likewise be convenient for subroutines compiled separately or in libraries; both CALLX and CALL may

not be needed for a given implementation. The *subname* and *xsubname* fields may have argument subfields in some implementations.

The *name* field may be any string of characters, but may not be used to designate passing of arguments.

CRISPFLOW produces the flowcharts below:



Translators encountering CALL and CALLX statements output appropriate SUBROUTINE linkages. The DO statement may generate a linkage to TO or PROCEDURE modules named by *name* within this compilation, or may substitute that module body directly inline into the target code stream (translated as per these specifications).

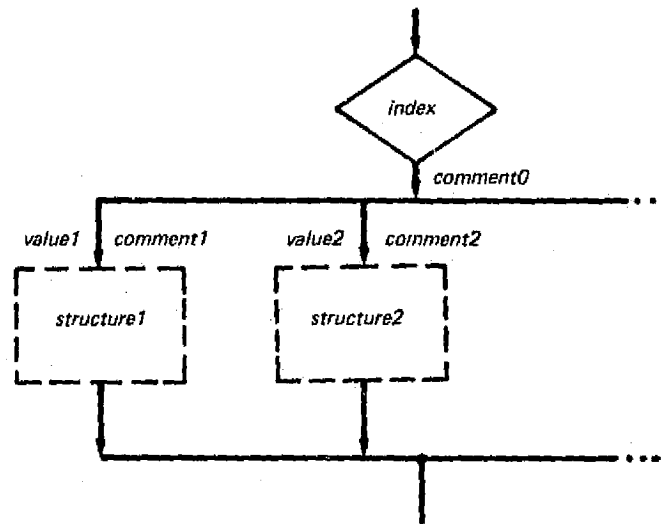
4.3.10 CASE Structure

The CASE structure implements a multiple branch on an index variable (or other discrete-valued variable).

```
CASE (index) [<*comment0*>]
:-> (value1) [<*comment1*>]
:   structure1
:-> (value2) [<*comment2*>]
:   structure2
:   ...
:   ...
... ENDCASES
```

The *valuei* fields represent valid values of the *index* and may appear in any order; several *values* may appear, separated by commas, in one parenthetical form. ELSE may appear in lieu of the final *value* (see Section 4.3.12). The *commenti* fields are optional, and are used only by CRISPFLOW to annotate flowcharts.

CRISPFLOW produces the following flowchart:



CRISP translators generate code to branch to appropriate blocks of code as specified by *values* of *index*. If an ELSE clause is present, all values of *index* other than those cited invoke the ELSE clause; otherwise, such values cause runtime error actions appropriate to the target language.

The CASE structure permits the selection of any one of several specified code bodies for execution, based on the value of a single scalar discrete-valued variable. Each case entry is labeled with the scalar value of the index required to cause execution of that case. These scalar labels need not be in consecutive order, nor must all possible discrete values in a range be present. Any discrete data type recognized by the target language as an ordered set is permissible, such as integers and characters. However, the pre-processor will not necessarily perform a check of the index data type versus the case-label scalar type, nor will all discrete data types permissible in a base language necessarily be supported by a CRISP translator.

The ELSE-case is a default specification, and is optional within the structure. If an ELSE-case is specified, then any value of the index not corresponding to a case label upon execution causes the ELSE code body (which may be null) to be executed. If no ELSE-case appears within the structure at all, then an index value outside the stated set of indices causes a runtime error message to be output on the job run stream and appropriate recovery action for the target language.

Several case labels may appear in the same clause header, to cause execution of the common case body for the specified values of the index. Commas separate the case labels in such instances.

Duplicate case labels cause translator diagnostic error messages to be printed.

4.3.11 DISPLAY, CANCEL, and ENABLE Statements

During the translation of a CRISP source into target statements, the statement

```
ENABLE MODULE COUNT
```

causes a set of special trace execution counters to be declared, one for each flowline. During execution of the target code, each traversal of a flowline causes incrementation of the corresponding counter.

The CRISP statement

```
DISPLAY THRU LEVEL integer
```

outputs target code that, when executed, prints the values accumulated in the counters. Each such count is tagged with the Dewey-decimal identifier of the corresponding program step. The *integer* value selects the module nesting depth of the display as determined by the decimal count in the Dewey-decimal.

The statement

```
CANCEL MODULE COUNT
```

disables the creation of flowline execution trace logic. The use of ENABLE-CANCEL pairs permits probing of selected portions of the program during development testing.

The default condition at the beginning of translation is the ENABLE mode. For fully verified programs, the overhead setting up and incrementing counters must be removed by prefixing the source program with the CANCEL statement.

4.3.12 ELSE Statement

The ELSE keyword may appear in IF or CASE structures to introduce the default clause. Parentheses are optional:

```
ELSE [<*comment*>]
(ELSE) [<*comment*>]
```

CRISPFLOW labels the ELSE flowline "F" in binary-IF structures and "ELSE" in multiple-IF and CASE structures. The *comment* field annotates this flowline additionally, if present.

Upon encountering ELSE, translators generate an unconditional transfer to the structure end, followed by a transfer label for the ELSE-clause.

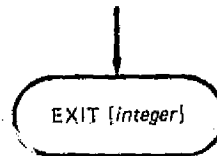
4.3.13 EXIT Statement

The paranormal EXIT statement may appear within the TO or PROCEDURE structure and the MACRO structure. The CRISP-PDL listing format is the same as for ABORT. In a TO-module, for example,

```
TO string
...
...
: <-----EXIT {integer}
:
:
:
ENDTO
```

The number of "-" which will appear is that required to fill the space from the indenting level of EXIT back out to the procedure first level. All subsequent lines have a colon (:) in column 12 and the final END will begin in column 12 as shown above.

CRISPFLOW draws the terminal symbol with EXIT *integer* enclosed, as



Translators set the OUTCOME flag to the value of *integer* and branch to the code generated by the module ender. EXIT may appear at any level of decision or loop nesting within the structure; EXIT may not, however, be used to escape from more than one nested macro or procedure module at a time.

4.3.14 FINISH Statement

The **FINISH** statement appears as the last line in a CRISP source, beginning in column 1, as

```
FINISH
```

All clean-up and operations required to terminate processing are performed in the program being processed as well as for the processor itself. No symbol is drawn in CRISPFLOW.

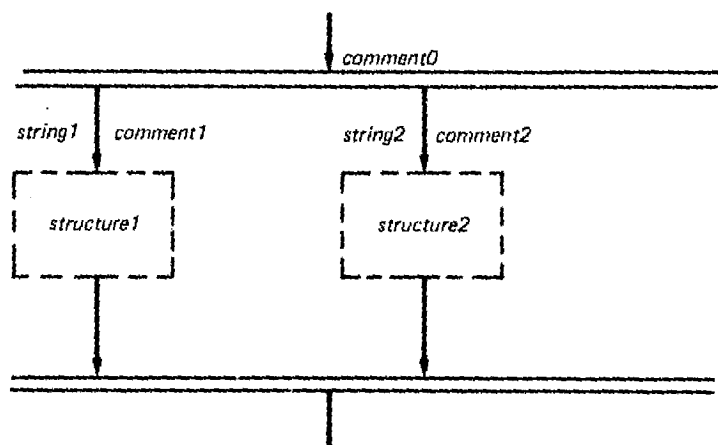
4.3.15 FORK Structure

All structures within this syntax are bounded at the top by **FORK** and at the bottom by **JOIN**. Within this construct, concurrent processes are identified by clauses introduced by parenthesized strings:

```
FORK [<*comment0*>]
:: -> (string1) [<*comment1*>]
::   structure1
:: -> (string2) [<*comment2*>]
::   structure2
::   ...
::   ...
:: ... JOIN
```

Any number of *structures* may appear, introduced by a parenthesized *string*.

CRISPFLOW produces the following flowchart:



FORK...JOIN will only appear in translators for languages that permit concurrent operations of the type inferred by the flowchart. *Strings* are necessary to label tasks for concurrent execution. JOIN terminates the concurrent mode for all of these tasks, and only when all have reached the JOIN.

4.3.16 FUNCTION Structure

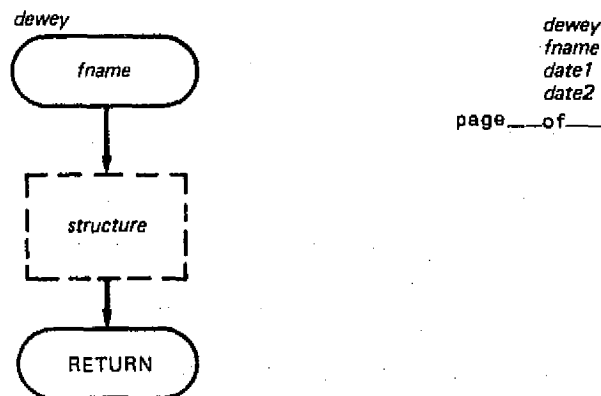
The FUNCTION structure permits definition of functional modules if supportable by the target language. If local and external functions must have different syntaxes, the keyword XFUNCTION may be added.

On input and listed output, FUNCTION begins in column 1 (see Section 4.3.5 for general header form):

```
FUNCTIONC fname [<*date1*>]                                [MOD# dewey]
      structure
      ENDFUNCTION [<*date2*>]
```

The ENDFUNCTION keyword normally appears listed in column 13, but moves to column 12 if an extra-normal exit (RETURN) has occurred in the module. The designator C denotes a colon (:) or space.

CRISPFLOW draws entry and exit terminal symbols labeled as shown below:



Functions are entities that are potentially separately compilable. Variables defined within a function are generally not accessible to the invoking program. Therefore, provisions for passing arguments or parameter lists are hereby specified. Functions may invoke subprocedures by using the DO statement as long as those procedures are defined as TO- or PROCEDURE-blocks within the function compilable unit. Functions may also invoke SUBROUTINES and other FUNCTIONS.

Each function name is a (unique) name to be assigned to the function in the target language syntax. Both the name and arguments are contained in *fname* and passed directly without analysis by the translator to the target language. The name and arguments must, therefore, conform to the conventions of the target language.

See RETURN (Section 4.3.27) for paranormal exit. Control flow reverts to the invoking program upon encountering the RETURN statement in the FUNCTION body, subject to the same rules as for subroutines.

Functions may be invoked by the usual base-language provisions for functions, if any. If none exists in the base language, then FUNCTION-block capability may be omitted from the translator.

4.3.17 IF Structure

The generalized IF structure permits the logical selection for execution of one among a number of specified code bodies within the structure:

```
IF (predicate1) [<*comment1*>]
:   structure1
:-> (predicate2) [<*comment2*>]
:   structure2
:   ...
:   ...
...ENDIF
```

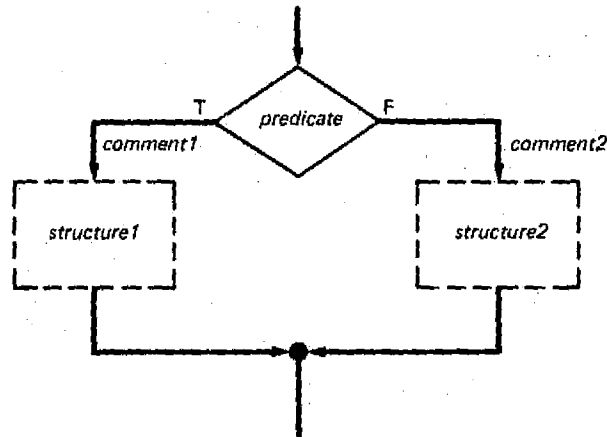
Any number of *structures* may appear, introduced by a parenthesized *predicate*. One ELSE or (ELSE) can also be used to introduce the final structure (see Section 4.3.12).

The *structure* (or clause) selected for execution corresponds to the first-encountered true condition in top-down order. All conditions tested have logical values *true* or *false*. Any of the selections may lead to null clauses, and the ELSE-clause is optional. If present, the ELSE-clause is executed as a default case when all of the given predicates are false. The ELSE keyword need not be stated when the code block following is null.

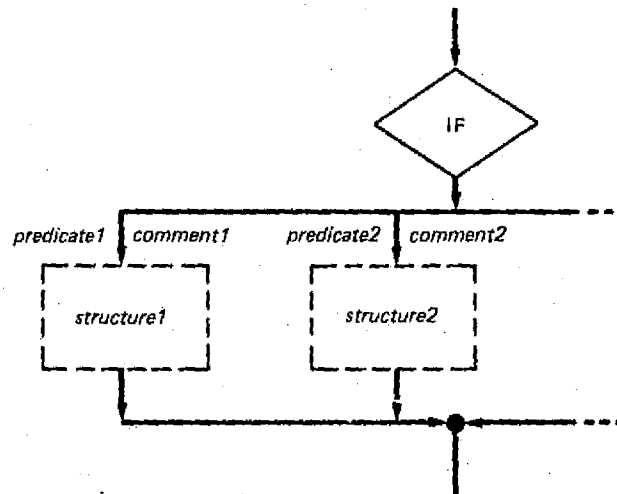
At the completion of execution of a given clause (as signalled by the appearance of a new clause or ENDIF), control passes to the end of the structure (designated by ENDIF).

There are no constraints on the number of predicates or on the size (including null) and number of clauses. The optional parentheses around ELSE are provided for consistency in appearance with other predicates, if the programmer desires.

CRISPFLOW distinguishes between the simple single-predicate form and the general form. The single predicate form (with an optional *comment2* on an ELSE statement) is



The general IF-form appears as a multiple branch:



4.3.18 IF Statement and Modifier

A special case of the IF-structure may be used in situations where there is a single predicate, the default case is null, and the code body to be executed when the predicate is true consists of a single statement. In such

344 Appendix C

cases, the single-line statement is placed on the same line as the IF keyword, and ENDIF is omitted:

```
IF (predicate) statement
```

The *statement* may be any single-line valid construction in the base language, or it may be a DO, CALL, or CALLX statement. The *statement* may not begin with a comment.

IF may also modify EXIT, LEAVE, CYCLE, and RETURN, as in the example

```
: <----EXIT integer IF (predicate)
```

Such a form is equivalent to

```
IF (predicate)  
:   EXIT  
...ENDIF
```

CRISPFLOW will produce the same flowchart as would be obtained from the corresponding longer form. If a box number and cross-reference field are designated, the box number is put on both symbols, but the cross-reference applies only to the *statement* box.

4.3.19 LET Statement

All identifiers signalled by the presence of periods and underscores are automatically catalogued, as previously described (Section 4.2.1.1). Other identifiers may be identified to the CRISP-PDL processor for subsequent cross-referencing purposes, using the statement

```
LET identifier BE string
```

The *identifier* is any alphanumeric (first character alphabetic), and *string* is any arbitrary set of characters (or may be null); *identifier* may contain periods and underscores, if desired.

Subsequent occurrences of *identifier* in the CRISP-PDL source program are treated the same as identifiers using periods and underscores. The *string* appears in the program glossary; it provides a means of defining variable name mnemonics, data type, units, and other pertinent information for later reference.

CRISPFLOW and CRISP translators ignore the LET...BE statement altogether. A LET statement without BE is always assumed by all processors to be a base-language statement, and no error is noted.

4.3.20 LOOP Structures

All loops in this syntax are bounded at the top and bottom by the verbs **LOOP** and **REPEAT**, respectively. Within this construct, the **LEAVE** verb may be used to cease iteration and to continue execution at the statement following **REPEAT**. Similarly, the **CYCLE** verb may be used to start the next iteration; that is, to transfer control to the repeat statement. Multiple **LEAVE** and **CYCLE** statements may appear within a given loop.

The format of the CRISP-PDL listing is

```

LOOP ...
! ...
! ...
! ...
!.. REPEAT...
```

There are three forms of valid loop structures, and variants within each form. There are two unindexed forms and one indexed form. These are

```

LOOP WHILE (predicate)
! structure
!.. REPEAT

LOOP
! structure
!.. REPEAT IF (predicate)

LOOP FOR index = expression1 [BY expression2] TO expression3
! structure
!.. REPEAT WITH NEXT index
```

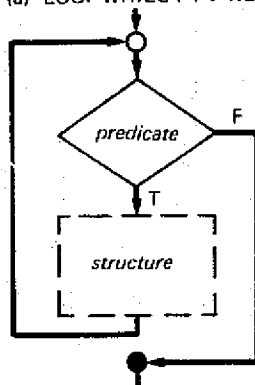
In the first of these, **WHILE** may be replaced by **UNTIL**, meaning **WHILE NOT**; in the second, **UNLESS** may replace **IF**, meaning **IF NOT**; and, in the third, a **WHILE** or **UNTIL** phrase may replace the **TO** phrase. All *expressions* are assumed to be in target syntax already. If *expression2* is omitted, a value of unity is assumed.

The *index* parameters must match in the third form. The value of the increment, *expression2*, is constant throughout the iteration, but may not be zero upon runtime entry into the loop; *expression3* is re-evaluated on each iteration of the loop only if **WHILE** (or **UNTIL**) is used.

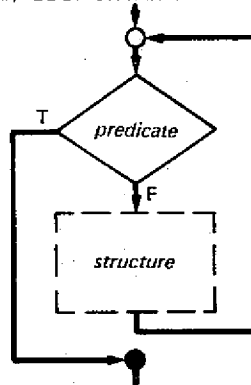
Upon termination of indexed iteration, the value of the *index* variable shall be the first such value encountered violating the loop predicate.

CRISPFLOW draws these structures as depicted in Figure 4.3.20.1.

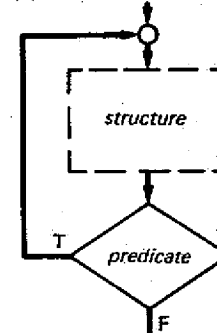
(a) LOOP WHILE . . . REPEAT



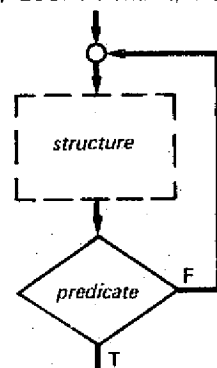
(b) LOOP UNTIL . . . REPEAT



(c) LOOP . . . REPEAT IF



(d) LOOP . . . REPEAT UNLESS



(e) LOOP FOR . . . TO . . .
...REPEAT WITH NEXT

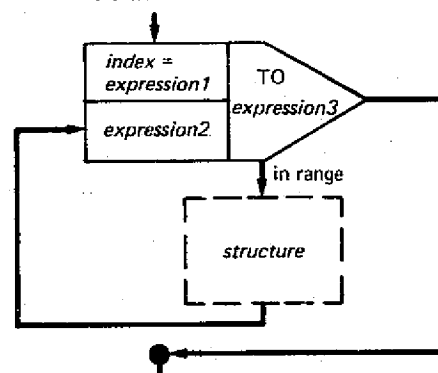


Figure 4.3.20.1. CRISPFLOW format of LOOP-REPEAT structures

The LEAVE statement may appear nested within a loop structure, as

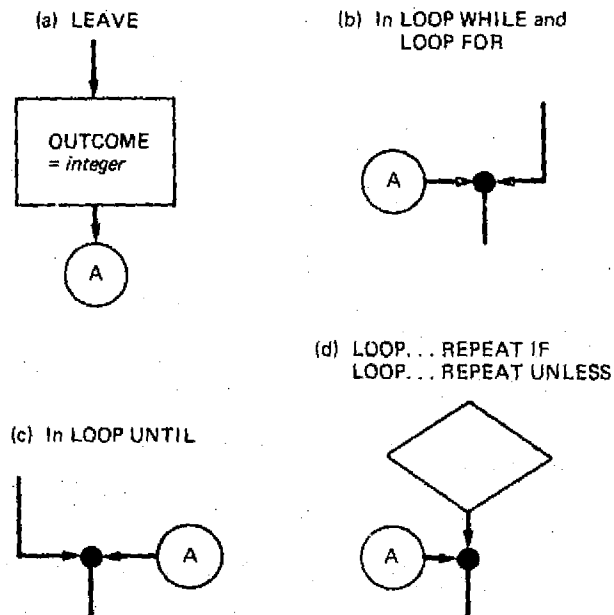
```

LOOP ...
  ! ...
  ! ...
  ! <-----LEAVE [integer]
  ! ...
  ! ...
  ! ... REPEAT ...

```

The number of dashes (-) which appear on CRISP-PDL listings is that required to fill in from the current indentation level, back out to the LOOP indentation level. Colons (:) appear on each subsequent line within the LOOP structure at the LOOP indentation level. LEAVE may only be used to escape iteration of the innermost loop structure containing it, and the LEAVE directive cannot be imbedded within procedures invoked by DO or CALL (unless LEAVE appears within appropriate LOOP-REPEAT structure boundaries within the procedure). If an *integer* follows LEAVE, its value is assigned to the OUTCOME flag when executed. When such is the case, normal loop termination sets OUTCOME to zero.

CRISPFLOW charts the LEAVE situation using on-page connectors labeled alphabetically (A, B, C, etc.) as shown below:



The CYCLE verb may also appear anywhere within the LOOP...REPEAT structure, and at any level of decision nesting within that range. CYCLE may only be used to control iteration of the innermost loop structure containing it. CYCLE may not, however, be imbedded in procedures initiated by DO or CALL (unless CYCLE appears within appropriate LOOP...REPEAT boundaries within the procedure).

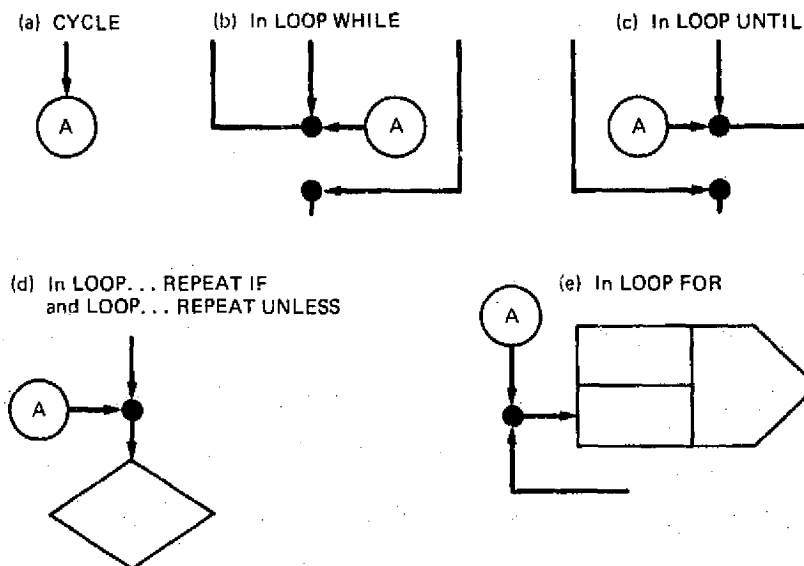
```

LOOP
! ...
! ...
! <-----CYCLE
! ...
! ...
! .. REPEAT

```

The number of "-" which will appear is that required to fill the space from the indenting level of CYCLE back out to the LOOP indenting level.

CRISPFLOW draws on-page connectors labeled alphabetically (A, B, C, etc.) for the CYCLE situation as shown below:



CRISP translators generate an unconditional branch to the beginning of the REPEAT statement for the CYCLE statement.

4.3.21 MACRO Structure

The MACRO structure defines a module for direct insertion. The MACRO keyword begins in column 1, as

```
MACROC template
      structure
ENDMACRO
```

See Section 4.3.5 for further details concerning module header format.

CRISP translators have a minimal, but useful, compile-time text-substitution macro capability. Macro invocations in the source program are signalled by the appearance of an escape character (%) followed by a string to be replaced:

```
...%string...
```

Such occurrences of *string* that "match" a macro *template* (matching described below) will be replaced by *structure* prior to generation of any code output.

The macro *template* consists of characters that must match corresponding characters in the input *string* exactly, except in positions in the *template* where markers for formal parameters occur. These markers take the form %1, %2, ..., %9; they designate the place in the input *string* where groups of characters are to be extracted and passed to the internal macro *structure* for substitution in places correspondingly marked %1, %2, ..., %9. The numbers of characters assigned to each parameter marker are determined so as to make all other characters in the input *string* agree with *template* characters. If no matching is possible, the %*string* is output directly as target syntax. See References 7.2.1 and 7.2.2 for further details on template matching based on the STAGE2 macro generator.

MACRO-defining modules and invocations may be used anywhere in the CRISP source code. In particular, an invocation may precede its macro module definition. Macro modules may contain other invocations, but may not enclose other macro modules (nor any other module types, for that matter).

Alternate forms of the macro module may be used to define macros within procedures, subroutines, functions, and programs. These are

```
%template MEANS string%END
```


and

```
%template MEANS
  structure
%END
```

In these forms, the *template* parameter markers are designated by only the escape character (% rather than %i); the *i*th such template marker is used exactly the same as %i in the MACRO template; up to 9 may appear. The *structure* (or *string* in the simple one-line form) uses the same conventions as in the MACRO *structure*, viz., %i.

CRISP-PDL and CRISPFLOW operate as if the separate MACRO modules were procedure modules; however, the alternate forms introduced by % are treated by CRISP-PDL as follows: The single-line form is listed at the current indentation level, and the multi-line form appears as

```
%template MEANS
:   structure
: ...%END
```

CRISPFLOW generates a separate one-page flowchart for each such macro definition.

See EXIT (see Section 4.3.13) for paranormal exit information.

4.3.22 NORMAL Statement

The NORMAL keyword may appear in AT or WHEN structures to introduce the clause that executes when contingency or priority event interrupts have not occurred. It may also occur in OUTCOME structures, in which case it corresponds to a zero value of the OUTCOME flag. Parentheses are optional:

```
NORMAL [<*comment*>]
(NORMAL) [<*comment*>]
```

For the AT structure, entry into any of its clauses (including NORMAL) disconnects the contingencies. For the WHEN structure, each clause identifier (including NORMAL) causes the translator to generate code to clear the interrupt logic of the event structure above it to return to the point of interruption, and to label the entry point to the current event clause.

The NORMAL clause need not appear in AT structures when it introduces a null clause, but must always appear in the WHEN structure, even with a null clause.

4.3.23 OUTCOME Flag and OUTCOME Structure

CRISP has a special internal flag which can be set only by RETURN, EXIT, LEAVE, and the normal module ender. Each of the paranormal exits has an optional integer field that is assigned to the OUTCOME flag on execution of the escape. Translators emit code for the normal module ender to clear the flag to zero if any paranormal escape has set the flag within the module. Similarly, if other paranormal exits occur unlabeled by an integer, they, too, cause the flag to be cleared. In this way, the OUTCOME flag may be used to record which of several paranormal escapes from a module or loop was taken.

This flag can only be tested, using the OUTCOME structure

```
OUTCOME [< *comment0* >]
: -> (value1) [< *comment1* >]
:   structure1
: -> (value2) [< *comment2* >]
:   structure2
:   ...
:   ...
... ENDCASES
```

The OUTCOME keyword is treated in all processors exactly as if CASE (OUTCOME) had appeared, except that the internal flag variable is referenced. NORMAL may replace a value of zero in a clause header.

4.3.24 PROCEDURE and TO Modules

The PROCEDURE and TO keywords introduce modules invoked by the DO verb. The two keywords are equivalent; ENDTO and ENDPROCEDURE are likewise equivalent, and any combination of header and ender pairs is permitted.

```
PROCEDURE name [< *date1* >]                                [MOD# dewey]
    structure
    ENDPROCEDURE [< *date2* >]

TO name [< *date1* >]                                       [MOD# dewey]
    structure
    ENDTO [< *date2* >]
```

Procedures, for the purposes of this specification, are bodies of code whose scope of variables is that of the module which has invoked that body

of code. In other words, the procedure has direct access to the set of variables defined in its hierarchic ancestors with which that procedure was compiled. No provision is made for passing parameters. (See Section 4.3.5 for generalized header information.) The *string* may consist of any input characters, but may not be used to designate passing of arguments. The *name* must be a valid identifier and must exactly match the field following only one DO somewhere in the same compilable section of CRISP source data in order to be translated correctly. *Name* is converted by CRISP translators to a unique program label.

CRISP-PDL identifies procedure modules unlinked to by DO and multiple DOs invoking the same procedure (not permitted) in the status report.

On execution, the ENDTO or ENDPROCEDURE statement signals a return of control back to the invoking module. Additional returns (or escapes) from procedures can also be effected by the use of the EXIT statement. See EXIT in Section 4.3.13 for paranormal exit information.

CRISPFLOW generates a one-page flowchart as illustrated in Figure 4.2.2.1 in response to procedure module source input.

4.3.25 PROGRAM Structure

The main program is announced by the PROGRAM statement. In base languages which require that this program have a named beginning, the program name must follow the conventions of the base language. Where naming is not required by the base language, the name may be ignored by the translator (in such cases, however, the PROGRAM statement must usually be the first executable statement).

The main program module continues until the ENDPROGRAM statement; statements past ENDPROGRAM (compiled with the main program) are generally subroutines, procedures invoked by TO and defined by TO (or PROCEDURE) structures, or else are data declarations or intermodule text.

The format is, therefore, of the form (see Section 4.3.5)

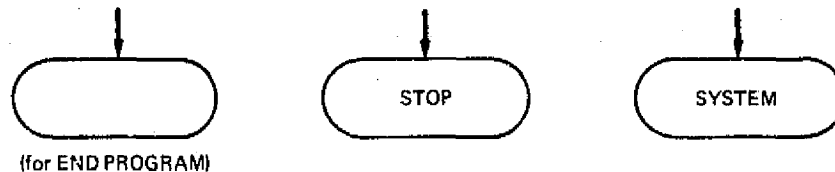
```
PROGRAMC pname [<*date1*>]                [MOD# dewey]
      structure
      ENDPROGRAM [<*date2*>]
```

Upon runtime execution of ENDPROGRAM, control flow returns to the system or processor controlling the execution of the program. (For example, batch FORTRAN probably returns to the operating system and thence to the user's job control stream. For interactive systems, such as BASIC, control probably returns to the BASIC command mode.)

To control the type of program exit when a choice is available, two directives, STOP and SYSTEM, are provided. STOP returns program control to a supervisory program controlling the program being executed, such as the BASIC processor mentioned above. SYSTEM returns control back to the operating system and to the user's job-control-language stream.

STOP and SYSTEM may only appear within the confines of PROGRAM...END-PROGRAM statements, and may not appear in subroutines, macros, functions, or procedures.

CRISP-PDL formats STOP and SYSTEM the same as ABORT (Section 4.3.7). CRISPFLOW generates the flowchart elements below:



4.3.26 REQUIRE Statement

The REQUIRE statement is of the form

REQUIRE AT *dewey*: *statement*

This causes the CRISP translator to extract and save *statement* during the first pass, and to insert it during the second pass into the object code immediately before the code for that statement whose *dewey* decimal identifier is given. This identifier is determined from the MOD# field of the appropriate module header and the step number *dd* within that module. See Section 7.2.4 of Reference 7.2.5 for further details.

4.3.27 RETURN Statement

The RETURN statement is a paranormal exit from a SUBROUTINE or FUNCTION module. The listing format is identical to EXIT, except for the keywords.

```
SUBROUTINE ...
...
...
: <-----RETURN [integer]
...
...
ENDSUBROUTINE
```

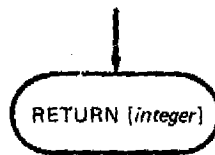
354 Appendix C

The action of RETURN is similar to that of EXIT. If an *integer* value is specified, that value is assumed to be assigned to the OUTCOME flag (Section 4.3.23).

CRISP translators then generate code to make such assignments and branch to the ENDSUBROUTINE (or ENDFUNCTION) point, which redirects control flow back to the calling program. ENDSUBROUTINE additionally resets OUTCOME to zero at a point above the return collection node if a RETURN *integer* has appeared.

RETURN may appear at any level of decision or loop nesting within the module; RETURN may not, however, be used to escape from more than one nested subroutine or function module at a time.

CRISPFLOW generates the terminal symbol:



4.3.28 STUB Identification

The STUB keyword is used only by CRISP-PDL and CRISPFLOW processors to identify procedures, subroutines, and functions that represent dummy structures to be removed later. The format is exemplified by the form

```
TO name
    STUB string
    ...
ENDTO
```

CRISP-PDL marks the *name* of the procedure as being a stub for display in the stub status report.

CRISPFLOW adds the word "STUB" to the upper right-hand module identification following the *name* and preceding the date fields:

```

dewey
name
STUB
date1
date2
page__of__
```

4.3.29 SUBROUTINE Structure

The subroutine, invoked by the CALL (or perhaps CALLX) statement, is the code body structure appearing between SUBROUTINE-ENDSUBROUTINE statements. Upon entry during execution, the first line after SUBROUTINE is the first executed; upon reaching the ENDSUBROUTINE statement, control returns to the calling program in the normal fashion.

Subroutines are potentially separately compiled entities. Local variables defined within a subroutine must not be accessible to the invoking program. Therefore, provisions for passing arguments or parameter lists are assumed. Subroutines may invoke subprocedures by using the DO statement, provided the invoked procedures are defined by TO or PROCEDURE structures in the same compilable unit. Subroutines may also CALL (or CALLX) other subroutines or invoke FUNCTIONS.

Except for the keywords, the listing structure of a SUBROUTINE is the same as any other module:

```
SUBROUTINE: subname [<*<date1*>>]                                [MOD# devery]
           structure
           ENDSUBROUTINE [<*<date2*>>]
```

See Section 4.3.5 for format and 4.3.27 for use of the RETURN statement to produce paranormal escape.

The *subname* contains the subroutine name and arguments passed to and from the subroutine. Base language syntax applies to this *subname*. If local and external subroutine definitions must have different syntaxes, the keyword XSUBROUTINE may be added.

CRISPFLOW generates the flowchart format illustrated in Figure 4.2.2.1, except that RETURN appears in termination symbols.

4.3.30 WHEN Structure

The WHEN structure is a means for handling priority interrupts for real-time processes. The syntactic form is illustrated by its CRISP-PDL listing form:

```

WHEN
::-> (event1) priority1
::   structure1
::-> (event2) priority2
::   structure2
::   ...
::-> (NORMAL)
::   structuren
... ENDWHEN

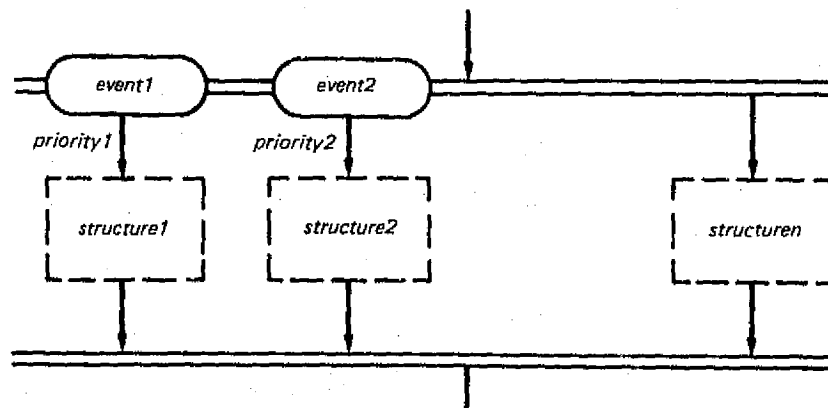
```

The CRISP translator generates code to connect and arm all priority interrupts for the *events* listed at the *priority* levels given (if permitted) and proceeds to execution of the NORMAL structure, which must be the last clause. Interrupts occurring during the execution of the NORMAL clause pass control to the corresponding *event* in order of priority. At completion of an event clause, control returns to the point of interruption. When the NORMAL clause completes at ENDWHEN, all interrupts for the events involved are disabled.

The *event* syntax is that of the base language. See Section 4.3.22 for further discussion of NORMAL.

In order for WHEN structures to be nested within other WHEN structures utilizing, perhaps, the same interrupt assignments, it may be necessary for WHEN...ENDWHEN to save and restore interrupt vectors or such other mechanisms used in the target language to maintain proper control correctness of program segments.

CRISPFLOW generates the following flowchart elements for the WHEN structure:



4.4 User/Operator Special Features

This specification does not address user/operator interfaces, procedures, or protocols. Inasmuch as user/operator I/O devices have been left unspecified, the form and extent of option selection is left open for system-dependent considerations.

However, users are assumed to be responsible for generating and editing the CRISP source input programs prior to invocation of the CRISP processors herein described.

In particular, CRISP program modules within a file are selectively processed by control data given to a CRISP processor of this specification. Users must first prepare alternate scratch files containing such selected modules if they wish to do so. The capability to extract and form such scratch files is not included in this specification of CRISP processors; however, such a capability is not proscribed by this document.

4.5 Data Base Specifications

CRISP-PDL and CRISP translators shall be designed in such a way that they may share a program listing file for display purposes (Section 4.3.11) in a future upgrade, so that target compiler error messages and execution statistics can appear on the cosmetized program source listing.

All CRISP-PDL report material and CRISP translator output shall be maintainable in compatible formats so as to be useful as data base elements in applications programs developments. Files shall be maintained in the formats specified by this document. All files written for retention by users shall be named by system-dependent standards that readily identify the file type (listing, target code, etc.) and the applications program to which they correspond.

5. PROGRAMMING SPECIFICATIONS

Programming specifications are not included in this version of this document.

6. TEST AND VERIFICATION SPECIFICATIONS

Test and verification specifications are not included in this version of this document.

7. APPENDICES

7.1 Glossary

ANSI. American National Standards Institute.

Base language. The language used in non-control statements in the source program processed by CRISP. Normally the same as or related to the target language.

CRISP. Control Restrictive Instructions for Structured Programming.

CRISP-PDL. CRISP Program Description Language.

CRISPFLOW. CRISP *FLOW*chart generating processor.

Dewey-decimal. A module identification scheme which concatenates, using a period separator, the current module Dewey-decimal with the step number within that module where a procedure is invoked. The Dewey-decimal so formed becomes the procedure identifier.

HIPO. Hierarchic Input, Processing, and Output. A graphical method of displaying functional specifications.

Module ender. One of the keywords *ENDTO*, *ENDPROGRAM*, *ENDPROCEDURE*, *ENDFUNCTION*, *ENDSUBROUTINE*, *ENDMACRO*, or *%END*.

Module header. The statement that introduces the definition of a *PROCEDURE*, *SUBROUTINE*, *FUNCTION*, *PROGRAM*, or *MACRO*.

Paranormal exit. An unstructured escape from a module or loop.

STAGE2. A general purpose macro processor based on template matching.

Target language. The language that a CRISP translator outputs in response to source input. Normally the same as the base language.

7.2 References

This appendix lists documents containing explanatory material referred to in other sections of this SSD.

1. Waite, W., "The Mobile Programming System STAGE2," *Commun. ACM*, Vol. 13, No. 7, pp. 415-421, July 1970.
2. Waite, W., *Implementing Software for Non-Numeric Applications*, Appendix A, Prentice-Hall, Inc., Englewood Cliffs, NJ, 1973.
3. *American National Standard Flowchart Symbols and Their Usage in Information Processing*, ANSI-X3.5-1970, American National Standards Institute, Inc., NY, Sept. 1, 1970.

- .4 Tausworthe, R. C., *Standardized Development of Computer Software*, Part II, Appendix B, this text.
- .5 *Ibid.*, Part I, Chapter 7, 1976.

7.3 Program Analyses

(To be supplied during program design phase.)

7.4 Sharable Subroutine Identification

(To be supplied during program design phase.)

7.5 Provisions for Future Modification

(To be provided during program design phase.)

7.6 Error Messages and Diagnostics

(To be provided during program design phase.)

7.7 Detailed Formats

7.7.1 Detailed Control/Response Message Formats

These formats are not applicable to this implementation-independent design, but will become applicable in an actual implementation. Standards for interactive and batch-mode operation will be established for each such application prior to such implementations.

7.7.2 Detailed Input/Output Formats

Input and output formats are permitted by this specification to be implementation-dependent only when conformance to this specification cannot be supported by the host system. Source input formats shall be compatible with the forms in Section 4.3. Further, the cosmetized reinput file output by CRISP-PDL shall also be compatible source input for CRISPFLOW and CRISP translators.

The remainder of this section covers output report formats for the CRISP-PDL processor as described in Section 4.2.1. All such outputs presume the availability of a 132-character-per-line output device.

7.7.2.1 CRISP-PDL Title Page Format

Figure 7.7.2.1.1 illustrates the format of the title page. Information on this page is taken from the first records input from the source medium when these records are textual (not module descriptions).

The first record is presumed to be a document number or other identifier; this record labels each remaining page of the report (centered on each page except the title page). The next set of source records, up to the beginning of a module or the ### signal, is copied directly onto the title page. The text signals ## and ### themselves are not printed on the title page. Title text may extend over more than one page if there are too many records in the first text block.

Titular material occupies the rightmost 80 columns of the page, the leftmost 52 being a CRISP logo, shown in Figure 7.7.2.1.2.

Source text records longer than 80 characters will be truncated on the right (an error will appear in such cases on the monitor device).

If the first block input from the source medium is a module, the title page is left blank (except for the logo), and a null (blank) line for the document identifier appears on subsequent pages of the report.

7.7.2.2 Table of Contents Listing Format

The table of contents is a listing of intermodule text lines signalled for printing by a leading #-space, together with names following module header keywords. These are printed in order of appearance and annotated with page and line numbers of occurrence within the CRISP-PDL listing. Each page of the table of contents contains a header with the program name (if any) and "TABLE OF CONTENTS" designation. Pages after the first add "(CONT.)". The format is shown in Figure 7.7.2.2.1. Page and line numbers may extend to four characters each, right-justified and zero-suppressed. No decimal appears in either.

7.7.2.3 Program Directory Listing Format

The program directory contains an alphabetic-order listing of all module names, status for each, and page and line of occurrence within the CRISP-PDL listing. The format is shown in Figure 7.7.2.3.1. The designation "(CONT.)" follows "DIRECTORY" on all but the first page. Page and line number specifications are the same as given in 7.7.2.2. The status field is six characters wide and contains the following:

"UNSEEN"	Name appears after DO or CALL, but does not appear on header
"NO CON"	Name appears on header, but does not appear after a DO or CALL
"STUB"	Module designated the name as a stub
" "	Otherwise

SSP-DSNSSP-002
ISSUE DATE _____

SOFTWARE SPECIFICATION DOCUMENT
DSN PROGRAMMING SYSTEM
CRISP PROGRAM DESCRIPTION LANGUAGE
DESIGN ANALYSIS PROCESSOR

PREPARER _____ DATE _____

APPROVED _____ DATE _____

CONCURRED _____ DATE _____

JET PROPULSION LABORATORY
CALIFORNIA INSTITUTE OF TECHNOLOGY
PASADENA, CALIFORNIA

Figure 7.7.2.1.1. Typical title page output format

```

CCCCCCCC
CCCCCCCC
CCC
CC
CC
CCC
CCCCCCCC
CCCCCCCC

RRRRRRRR
RRRRRRRR
RR   RR
RRRRRRRR
RRRRRRRR
RR   RR
RR   RR
RR   RR

IIII
II
II
II
II
II
II
II
IIII

SSSSSSSS
SSSSSSSS
SSS
SSSSSSSS
SSSSSSSS
SSS
SSSSSSSS
SSSSSSSS

PPPPPPPP
PPPPPPPP
PP   PP
PPPPPPPP
PPPPPPPP
PP
PP
PP

JET PROPULSION LABORATORY
4800 OAK GROVE DRIVE
PASADENA, CA, 91104

```

Figure 7.7.2.1.2. CRISP-PDL logo format

Figure 7.7.2.2.1. Table of contents format

[illegible]

Figure 7.7.2.3.1. Program directory format

7.7.2.4 Tier Chart Listing Format

The tier chart lists module names in program invocation-tree (control nesting) order, along with a status code followed by the line and page of occurrence. Status and occurrence printing are covered by 7.7.2.3. Names are indented as specified in Figure 7.7.2.4.1 to indicate the invocation level of nesting within the program. The main program and all subroutines, functions, and block macros appear at level 1. Procedures and subroutines invoked by DO and CALL (X) within a module are listed below the name of that module, indented as shown.

Module names are truncated on the right if indentation interferes with status printing (an error message also appears on the monitor device).

"(CONT.)" is added after "TIER CHART" on all but the first tier chart page.

7.7.2.5 Module Status Report Format

This report contains a list of names, status, and locations of modules having status "STUB" and "UNSEEN." The report format appears in Figure 7.7.2.5.1. "(CONT.)" appears on all pages but the first. Page and line numbers are covered by 7.7.2.2. The fields *nnnn* and *mm* are zero-suppressed, right-justified, with no decimal.

"Unseen" module names are counted and listed only once, even though there may be multiple occurrences in the procedural listing; the page and line numbers are, thus, only typical invocations. (The cross-reference concordance contains all such occurrences, however.)

7.7.2.6 Cosmetics and Indentation Format for CRISP-PDL

This section details the output procedural listing format of the CRISP-PDL structures. Each page of the listing contains the document number, program name, source file name, and current page number. Each line begins with the source file *line* (record) number, up to four characters zero-suppressed, no decimal, right-justified, and followed by two spaces, as shown in Figure 7.7.2.6.1. The remaining 126 character positions on the line are formatted as specified in Section 4.3. Column positions referred to in Section 4.3 are relative to the first character of this 126-character format (i.e., "column 1" or "left margin" in Section 4.3 refers to character position 7).

7.7.2.7 Glossary and Cross-Reference Table Listing Format

The glossary and cross-reference table reports each identifier used in the program in lexicographic order in the format shown in Figure 7.7.2.7.1. "(CONT.)" follows "GLOSSARY" on all but the first page. The *page* number at the bottom is the page number of the current page of the report. The

document number

program name

PROGRAM TIER CHART

NESTING

STATUS:LINE:PAGE

```

program . . . . . status:line:page
: module name . . . . . status:line:page
: : module name . . . . . status:line:page
: : module name . . . . . status:line:page
: module name . . . . . status:line:page
subroutine name . . . . . status:line:page

```

```

: : : : module name . . . . . status:line:page

```

roman numeral

Figure 7.7.2.4.1. Tier chart listing format

document number
program name

M O D U L E S T A T U S

SUMMARY:
TOTAL MODULES IDENTIFIED: *nnnn* 100%
NON-STUB MODULES: *nnnn* *nnn*%
STUB-STATUS MODULES: *nnnn* *nnn*%
IDENTIFIED, BUT NOT SEEN: *nnnn* *nnn*%

MODULES IN STUB STATUS:
module name *line:page*
.
module name *line:page*
.

MODULES IDENTIFIED BUT NOT PROCESSED:
module name *line:page*
.
module name *line:page*
.

roman numeral

Figure 7.7.2.5.1. The stub-status report

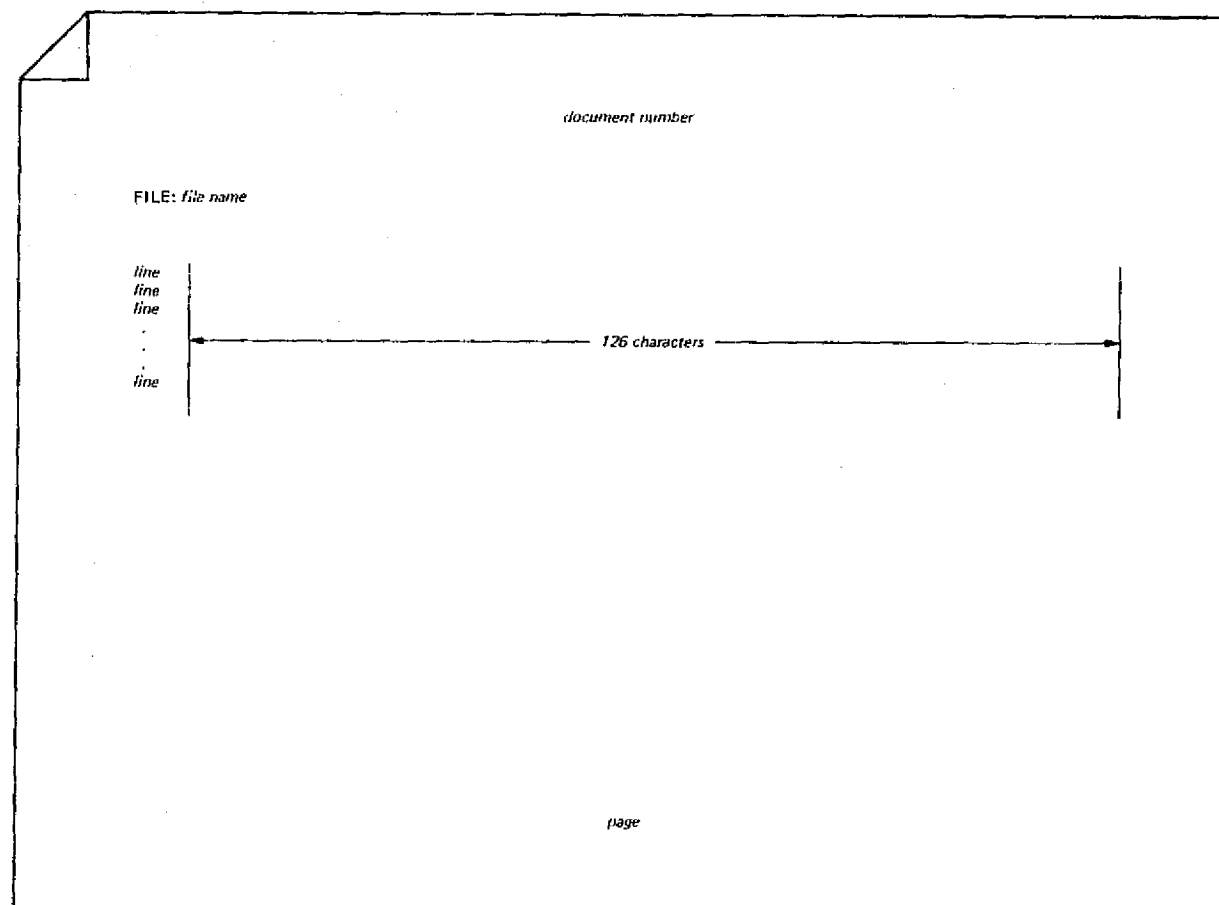


Figure 7.7.2.6.1. Page format of procedure and intermodule text listings

name field extends to 32 characters for each identifier. The *type* designator begins immediately (one space) after the *name* and is one of the following:

(PROGRAM NAME):
 (PROCEDURE NAME):
 (SUBROUTINE NAME):
 (FUNCTION NAME):
 (MACRO NAME):
 (IDENTIFIER):

After one more space, the *text definition*, if any appears in a corresponding LET...BE...statement, follows. The *line: page* designation on the *name* line gives the line and page numbers of the LET...BE...statement (this is absent if the identifier had no LET...BE...). Beneath each *name* line, the *line: page* designators list all the locations of appearances of that identifier.

7.7.2.8. Statistical Summary Format

The statistical summary format sketched in Figure 7.7.2.8.1 permits printing of the numbers of occurrences of keywords, module source lines, intermodule text lines, histograms of module and text block lengths, and other statistical data, as specified below.

The statistics printed under "NUMBER OF SOURCE LINES" are

IN MODULES:

NO. OF MODULES: *nnnn*
 TOTAL LINES: *nnnn mm%*
 COMMENT STATEMENTS: *nnnn mm%*
 UNCOMMENTED STATEMENTS: *nnnn mm%*
 MIXED STATEMENTS: *nnnn mm%*
 TOTAL STATEMENTS: *nnnn=100%*
 CONTROL STATEMENTS: *nnnn mm%*
 NON-CONTROL STATEMENTS: *nnnn mm%*
 AVG LINES/STATEMENT: *n.n*
 AVG STATEMENTS/MODULE: *nn.n*
 AVG LINES/MODULE: *nn.n*

IN TEXT BLOCKS:

NO. OF BLOCKS: *nnnn*
 TOTAL LINES: *nnnn mm%*
 AVG LINES/BLOCK: *nn.n*

Fields designated as *nnnn* or *mm*, above, represent numeric 4- and 2-character width outputs, right-justified, zero-suppressed, no decimal. Other designations with decimal are as shown. Comment statements are classified as any statement with '<*' in the keyword position.

document number				
G L O S S A R Y				
name	type	text definition	line:page	
IN module name, PAGE		page		
	line	line	line	
.				
name	type	text definition	line:page	
IN module name, PAGE		page		
	line			
.				
.				
page				

Figure 7.7.2.7.1. Glossary format

document number

program name

STATISTICAL SUMMARY

NUMBER OF SOURCE LINES: norm=100%

...

...

...

LENGTH STATISTICS:

...

...

...

KEYWORD, MODULE NAME, AND IDENTIFIER REFERENCES:

...

...

...

page

Figure 7.7.2.8.1. Format of the statistical summary

372 Appendix C

Within the "LENGTH STATISTICS" entry are printed histograms of the form

MODULES	INTERMODULE TEXT
1-5:xx	1-5:
6-10:xxx	6-10:r
11-15:xxx	11-15:xx
16-20:r	16-20:xxx
21-25:	21-25:xx
26-30:	26-30:r
31-35:	31-35:
36-40:	36-40:
OVER 40:	OVER 40:

The x designators in the histogram denote that characters will be printed, nominally one for each five lines counted; if fewer than five appear at the last place, "A," "B," "C," or "D" appears. The designations are

A=1
B=2
C=3
D=4
E=5

In computing text block length statistics, ##, ###, and #n signals are not counted, but do add to the total source line count.

Under "KEYWORD, etc.," the following statistics appear:

TOTAL REFERENCES: nnnn=100%
MODULE REFERENCES: nnnn mm%
IDENTIFIER REFERENCES: nnnn mm%
IN TEXT: nnnn mm%
IN STATEMENTS: nnnn mm%

KEYWORD OCCURRENCES: nnnn=100%
ABORT nnnn mm
.
.
.

Each of the CRISP keywords is printed in alphabetic order in columnar format across the page.

END OF SPECIFICATION

APPENDIX H

DEVELOPMENT PROJECT NOTEBOOK CONTENTS

The Project Notebook, described in outline form in this appendix, is primarily an administrative and management tool. Pertinent technical information regarding software design, testing, operation, and use appears principally in other documents outlined in this set of appendices. Instead, the Project Notebook described herein provides a centralized record of current and archived material related to the production of that software.

The Project Notebook is an informal, working-level document: handwritten notes, typewritten memos, etc. The arrangement of material suggested below ranges through the most useful project management information, in terms of aiding the project manager in his function. Figure H-1 is a top-level view of the notebook organization.

The primary purpose of the notebook is visibility into the current development status. However, some of the items included in it (e.g., the work breakdown structure) are more directly related to estimations of manpower, resources, and schedules; others (e.g., memos, minutes, decision log) record data that may be needed for reference later in the project; still others (e.g., change control log, standards waivers) are project control mechanisms.

At project completion, the archived material serves as a means for assessing overall productivity figures, costs, and the utilization of resources; it also provides useful information that can guide the formation of better methodology standards and more accurate scheduling and costing methods.

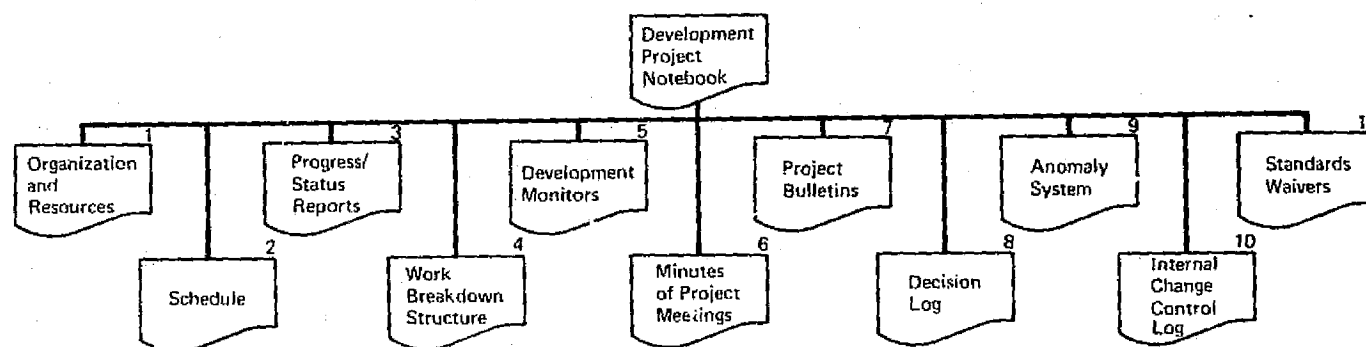


Figure H-1. Suggested organization of the Development Project Notebook

In large projects, the notebook may well require several volumes, perhaps one topic below for each. In smaller projects, a loose-leaf binder with index-tabbed separators may suffice.

If facilities and resources are available, much of the material may be kept and maintained in computer files (e.g., the tier chart and work breakdown structure). In any case, the material must be conveniently at hand for viewability on demand by the project manager or others, and must record the information in an easily usable form.

DEVELOPMENT PROJECT NOTEBOOK

Suggested Contents

FRONT MATTER

Title Page. Provide a title page containing (1) project identification number; (2) program, project, subsystem, and system titles; (3) PROJECT NOTEBOOK; and (4) name and organization of custodian.

Directory. List the sections of the Project Notebook by volume number (or file identifier). This directory is probably not necessary if the topics all fit in one notebook (or file drawer) with appropriately tabbed indices.

CONTENTS OF PROJECT NOTEBOOK

1. ORGANIZATION AND RESOURCES

Provide an organization chart and role statements as needed. Identify the manpower, dollars, and other resources available, and those needed to complete the development. Display a profile of these resources as a function of time.

2. SCHEDULE

2.1 Current Schedule

Provide a current high-level "master" schedule to show the committed baseline, controlled milestones (e.g., project review dates), and supporting major achievements during the development. Maintain current lower-level schedules keyed to work breakdown structure tasks identified and described in Section 4, below. Date each page of the schedule. For each line item, show start of work and the period(s) thereafter until milestone achievement. Distinguish unachieved from achieved milestones. For example, designate original milestones as unfilled deltas, to be filled in when achieved. Distinguish slipped milestones, say, as inverted deltas on the same schedule line. Identify work periods as, for example, horizontal parallel lines from start of work (broken, if there are periods of inactivity) to the delta. Darken between these lines to indicate progress.

2.2 Significant Notes

Record reasons for current slippages (unless contained in progress reports archived in Section 3, below), assumed contingencies, and other schedule-related matters as needed to indicate the current plan or to facilitate project operations.

2.3 Schedule and Note Archives

Retain outdated project schedules and schedule notes for reference needed later during this project, as well as for historical purposes.

3. PROGRESS/STATUS REPORTS

Insert the regular progress reports written by project members to project management, or by project management to higher management, the customer, or other organizations. Retain past reports for later reference during this project, as well as for historical purposes.

Such status reports should record (1) significant milestones achieved; (2) change in development plan and rationale for change; (3) resource expenditures; (4) activity; (5) forecast of activities; (6) computer utilization figures, such as computation costs, numbers of debugging runs, etc.; (7) progress monitors, such as percentage of total milestones achieved, pages of documentation, lines of code, etc., generated; (8) problems and potential problems; (9) other information "for the record."

4. WORK BREAKDOWN STRUCTURE

The work breakdown structure (WBS) contains necessary information to provide visibility and control, using quantifiable relationships and interdependencies of resources, schedule, and technical performances.

4.1 Task Hierarchy

Organize and display the software development effort as a tree hierarchy of work packages (tasks) that are significant, finite, and manageable, with quantifiable inputs, outputs, resources, schedules, and assigned responsibilities. Each task should ideally require the same effort.

4.2 Task Descriptions

Describe each task in the WBS in a uniform format. Include, as a minimum, the following: (1) date the description form was completed; (2) revision number of the description; (3) WBS task title and Dewey-decimal identifier in the WBS hierarchy; (4) responsible individual; (5) duration, planned and actual; (6) task description, or brief identification of what the task is intended to accomplish, what the task is part of, etc.; (7) schedule with start and finish dates, including appropriate milestones for task

elements; (8) task budget or time-phased manpower by skills categories necessary for task accomplishment, including supervision, design, coding, testing, documentation, and support services; (9) task scope, or estimate of lines of code, number of flowcharts and narrative, etc.; (10) inputs required, such as documentation, other task outputs, hardware, or other resources that are necessary to accomplish task, including need dates for each; (11) task outputs, interim and final, such as reports, supporting documentation, design documentation, operating code, listings, etc., including output dates for each; and (12) task interfaces, such as common software, interchangeable modules, or other relationships between this and other tasks.

4.3 WBS Integration

Display task interdependencies in some suitable way, such as PERT or schedule networks (cost or time). Identify critical-path tasks and float times.

4.4 WBS Archives

Retain previous WBS task descriptions, etc., after updates for historical purposes.

5. DEVELOPMENT MONITORS

5.1 Tier Chart

If not contained in Section 4, above, maintain a current tier chart of the developing program. Identify modules by category for status reporting. Such categories might include: (1) module approved into project change control; (2) module completed, but pending project approval into internal change control, or being reworked for reapproval; (3) module exists in preliminary or look-ahead form in SDL; and (4) module identified in program tree but not yet seen by SDL. These categories apply to design items, documentation items, coding, and testing.

5.2 Production Log

Maintain a log or set of logs that summarize the current production status and disposition of resources and materials. This part of the notebook should record (1) the traffic and custodianship of documentation items, data, program tapes, and disk files; (2) resources expended (manpower, CPU usage, number of debugging runs, etc); (3) implementation statistics (numbers of lines of code entered, altered, etc.); and (4) tests conducted.

5.3 Rate Charts

Maintain graphs of cumulative activities as a function of project day, suitable for forecasting future accomplishments, identifying trouble spots, calibrating productivity, etc. Such plots should display cumulative milestones, modules completed, number of anomalies discovered and repaired, schedule days, and manpower.

6. MINUTES OF PROJECT MEETINGS

6.1 Project Meeting Notes

Record the minutes of all project meetings. Give the date, attendance, agenda, and pertinent facts or issues discussed. Identify action items and responsibilities.

6.2 Action Item Log

Maintain a current list of all action items, the responsible parties, assigned compliance dates, and disposition.

7. PROJECT BULLETINS

Retain all project bulletins and other pertinent memoranda needed for project reference or historical purposes, issued within or received from sources external to the project.

8. DECISION LOG

Record project decisions that bear on later project plans and for historical purposes. Decisions which may be useful to or influence program sustaining and maintenance may be recorded here for convenience; however, these latter decisions should also be inserted eventually into Appendix 7.10 of the SSD.

9. ANOMALY SYSTEM

9.1 Anomaly Status Log

Maintain a list of all reported anomalies as described in 9.3, below. Give for each: (1) anomaly number; (2) short problem description; (3) system/subsystem/program identifier; (4) discovery date; (5) priority category; (6) assigned responsibility; (7) required closure date; (8) actual closure date; (9) disposition method and action.

Display graphs of the number of anomalies discovered and number of anomalies cleared versus date or usage time (either here or in Section 5.3).

9.2 Anomaly Descriptions

Retain copies of all individual anomaly forms summarized in 9.1, above, in this section.

9.3 Anomaly Definition

Identify or give a reference to criteria for anomaly detection and reporting during the production phase (including acceptance testing). Define priority categories, such as: (A) removal of anomaly is critical for usage of the software as required; (B) anomaly degrades performance or increases operational risks; and (C) anomaly does not prevent software from being used successfully, but is undesirable in that it requires some user/operator reorientation or work-around procedure.

9.4 Anomaly Reporting System

Describe (or reference) the anomaly reporting system, including roles of individuals involved with supervision, testing, correction, QA, etc. Also, describe any pertinent interfaces and procedures that are used.

10. INTERNAL CHANGE CONTROL LOG

10.1 Change Status Log

Maintain a summary list of deliverable items (modules, documentation, code, etc.) in conjunction with the tier chart in Section 5, above, or the WBS in Section 4, above, that records the traffic during development, so as to keep track of development items among subcontractors, team members, QA, etc.

10.2 Change Control Log

Retain copies of all development change-request/change-order forms in this section. Such request forms should contain (1) name of requester; (2) date; (3) system/subsystem/program/module identifier; (4) reason for proposed change; (5) description of proposed change; (6) seriousness; (7) method of change; (8) estimated resources required; (9) analysis and remarks; (10) availability of required resources; (11) constraints on resources; (12) related changes requested, implemented, or to come; (13) need date; (14) priority; (15) approval or denial and reason.

10.3 Change Control System

Identify or give a reference to criteria for change control action during the software development phase. Describe (or reference) policies and procedures that effect such changes, and include roles and interfaces of

individuals involved with supervision, design, coding, testing, documentation, etc.

11. STANDARDS WAIVERS

11.1 Standards Waiver Policy and Procedure

Record (or reference) the policies and procedures regarding waiving of standard practices during software development, including required levels of authorization for the various standards imposed.

11.2 Standard Waiver Log

Retain copies of all Standards Waiver Request forms in this section. Each waiver request should contain (1) name of requester; (2) system/subsystem/program/module name; (3) date; (4) standard to be waived; (5) scope of waiver; (6) reason for request; (7) description of alternative to standard and justification for use; (8) approval or denial.

REMOVING PAGE BLANK NOT FINISHED

APPENDIX I

OPERATIONS MANUAL CONTENTS

This appendix contains an outline of topics typically considered for inclusion in a Software Operations Manual (SOM). The items listed are not exhaustive, nor are all of those given necessarily applicable to a particular given operator guide. Rather, the topics herein contained are those that should be considered as candidates for inclusion in such a guide.

As in all documentation, the preparer must address his material toward a set of intended readers. For operational use, this orientation should primarily be toward the operator during an actual run of the program. Secondly, the manual should serve as a training base for operators. Finally, it should serve as an information medium to those who would read the manual to survey the appropriateness and adequacy of the program for subsequent operational activity. The outline below is an attempt at providing a logically and hierarchically arranged checklist.

This text has repeatedly recommended that manuals be written at least in a skeletal form from the top down (in detail hierarchy) concurrently with the writing of the SSD hierarchy and with the construction of the program, so as to provide timely information among developers, to permit the operator manual to be tested concurrently with the program, and to avoid last-minute efforts to complete the documentation prior to software delivery. The emphasis in writing the operator manual is on providing complete and effective information for exercising all of the options and operational functions of the program. The timely gathering of information and writing of technical material for the manual, however, must not be put in series with the formal, more clerical aspects (such as typing and reproduction) of a documentation activity.

As the program construction proceeds in a top-down manner, operational information in greater and greater detail typically becomes available. If

compiled and written into the guide during this time, the information level will tend to aid in assessing whether the emerging program falls within its operational requirements implemented so far.

Figure I-1 is a top-level view of the suggested document organization; greater hierarchic detail is provided in the written outline that follows. This outline contains guidelines after each topical heading for the type of material to be inserted at that point. In full, the topics constitute Class A detail.

As presented here, the operator is not assumed to be the user of the program; therefore, there are no sections devoted to applications. When operations personnel are intended to perform some of the functions of users, the topical outline in Appendix F can serve as a guide for selecting such material as appropriate for inclusion in the manual.

The operator typically receives requests, instructions, data and parameters for running the program and for performing other services having to do with interfacing the program capabilities with applications and system-level personnel. This often includes such things as scheduling of runs and resources, maintenance of tapes and files, loading of program elements and data, policing authorized use of the program and data, monitoring execution, recording performance characteristics (e.g., costs, number of users, anomalies, etc.), recognizing and reacting to alarms, and routing of the program outputs.

In orienting the material toward the intended operators, use the "robotic" instructional approach. Make the manual "stand-alone," extracting necessary operational procedures from existing references if needed to guarantee stand-alone operation. Other references, supporting narrative, examples, etc., may be included as necessary, and are encouraged.

In generating an SOM to the outline attached, arrange material global to a set of subsections under the introductory superheading for those subsections. Use short instructions with specific examples that actually run. If syntactic variables are used to illustrate a generalized form of input or output, clearly display and explain the syntactic convention and the valid substitution values of each syntactic variable. Then give valid examples on the use of the form being described.

Keep the manual as short as possible while making it say what needs to be said. Strive for clarity and completeness in exposing the material, but tempered by conciseness.

Sources for the material in this appendix are [5] and [44] through [46].

05

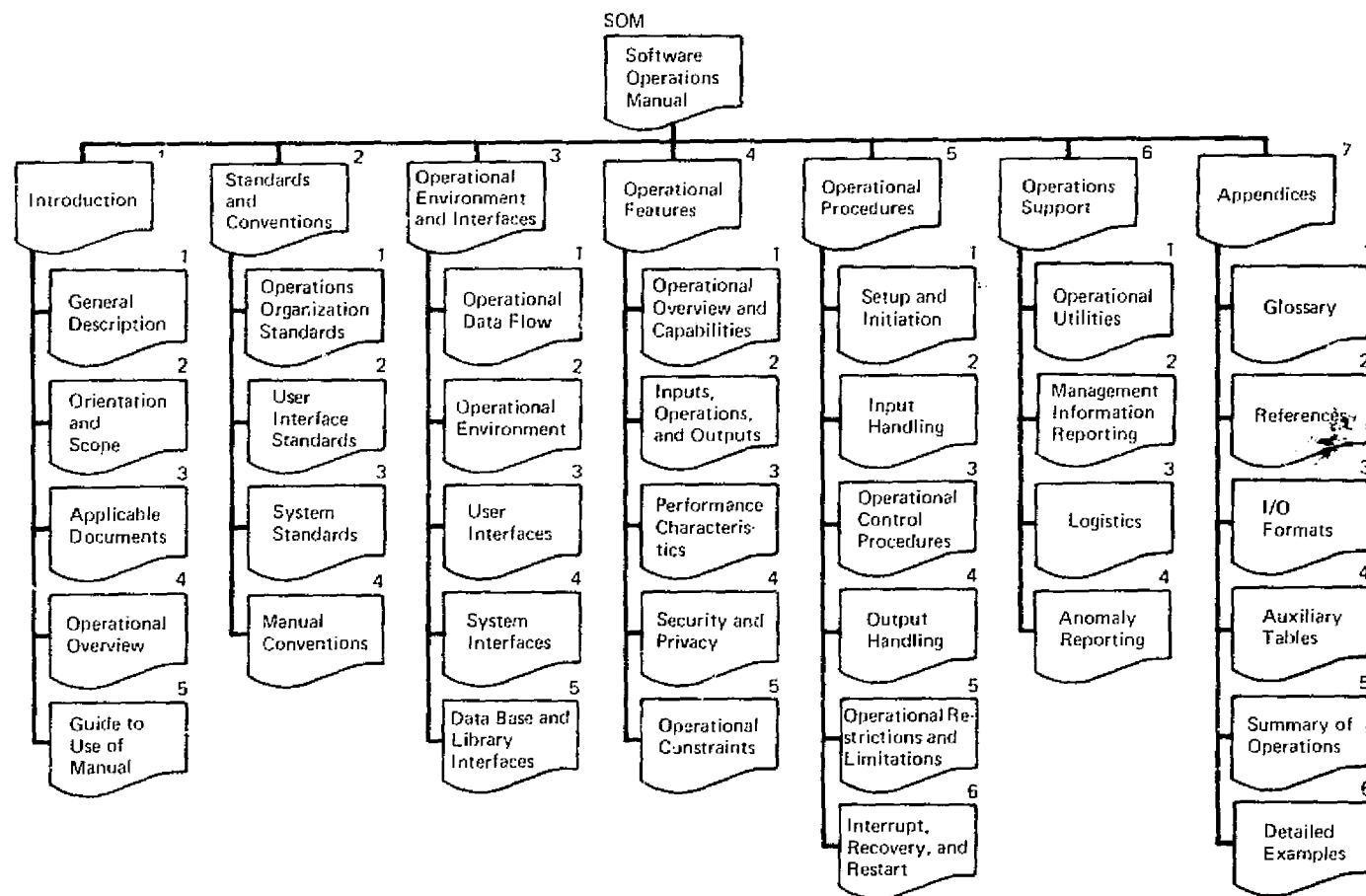


Figure I-1. Graphical outline of the SOM

OPERATIONS MANUAL

Detailed Table of Contents

FRONT MATTER

Title Page. Provide a title page containing (1) document number; (2) program, subsystem, and system titles; (3) SOFTWARE OPERATIONS MANUAL; (4) publication date; (5) author and management authority signatures, as appropriate, and (6) publishing organization. Signature or publication approval should be supplied only at SSD completion. The date reflects the time of latest change to any item in the manual.

Abstract. Give a brief abstract that summarizes the purpose and usage of this manual.

Change Control Information. Provide a statement that specifies the current level of change control authority, and describe procedures for submitting change requests and reporting anomalies.

Distribution Information. Provide information that tells how copies of this document may be obtained.

Table of Contents. Provide a detailed table of contents for the manual, which lists section number, title, and page of every item with a heading. (This is probably the last-supplied item in generating the manual.)

TEXT OF MANUAL

1. INTRODUCTION

1.1 General Description of the Manual and Its Use

Describe the purpose of the manual and the salient features involved with its operation. Summarize the specific uses of the manual, to whom it is addressed, why it is necessary, how and when it is intended to be used, the necessary level of operator expertise, and, finally, where one may go to obtain additional relevant material.

Also appropriate in this introduction are background information, history, relationships to other programs or system, and other operational matters.

1.2 Orientation and Scope

Summarize the extent of the material given, the range of application of the program, the operational personnel required, and major constraints in operating the program.

1.3 Applicable Documents

Identify all documents, controlling or informational, which apply to, regulate, or extend the operation of the program. Include a brief statement of the content of each document, its type (e.g., SSD), and purpose.

1.4 Operational Overview

Introduce the operational environment, the program function, and the type and content of data input, processed, generated, or transmitted. Describe the operator role with respect to usage of the program. Identify the salient characteristics of the program, such as: (1) real-time, interactive or batch; (2) computational or data manipulative; (3) developmental or operational. Identify the general system or subsystem environment (hardware and software) in which the program operates (a block diagram or data-flow diagram is useful here). Leave details, however, to ensuing sections of the manual.

1.5 Guide to the Use of the Manual

Explain how this guide is to be used in operating the program.

2. STANDARDS AND CONVENTIONS

This section describes standards imposed on or by the operating organization, and conventions (e.g., notations and terminology) applied in the manual.

2.1 Operations Organization Standards

Identify or reference applicable existing organizational standard operating procedures, state any exceptions to these standards necessitated by operational considerations, and provide any special standards required for effective operation of the program.

2.2 User Interface Standards

Identify or reference applicable existing procedural or liaison standards pertaining to the user/operations interface, state any exceptions to these standards negotiated as required for usage or operation, and provide any special standards required for operators to interface properly with users.

2.3 System Standards

Identify or reference applicable existing system standards relative to operational interfaces, state any exceptions to these standards, and describe any special standards pertaining to the system.

2.4 Manual Conventions

Define notations, terms, and other conventions or assumptions used generally throughout the manual. Include such items as ways of distinguishing literal fields from syntactic variables in descriptions of operator input and output formats, non-standard mathematical usage, special acronyms, etc.

3. OPERATIONAL ENVIRONMENT AND INTERFACES

This section describes the operational aspects of the program that highlights man (operator)/machine/system/software interfaces. This section is not procedural but environmental.

3.1 Operational Data Flow

Describe the data flow from user(s) to operator(s), through the system and the program, and back to the user(s). Describe the role of operators in effecting this flow. Identify operational interfaces with users, the system, and library and support facilities.

3.2 Operational Environment

Introduce the general environment within which operators interface with the users, the system, other software, and with the program. This section should focus on the operational environment in both static and dynamic terms. The static portion of the environment is defined by the relationships between the program and its interfacing with the operator, the system in which it is imbedded, and, perhaps, other systems with which it communicates. Data flow and operational modes describing system/subsystem/operator interactions and sequences define the dynamic environment.

Identify (1) interfaces among operators (if any); (2) the location of operators; (3) the sources of operational data; (4) the media used by operators to control execution, input data, and receive output; (5) manual tasks, etc. Describe operational interfaces with management, if appropriate. Defer operational features and procedures, specific formats, units, etc., until later sections.

3.3 User Interfaces

Identify and describe the interfaces between operational environment and the program user community. Discuss, as appropriate, forms, interfacing procedures, offline storage media, data input methods, modes of delivery of output to users, manual tasks, etc. Identify those items that are unique to this program and not covered by an overall system description or governing document (if this manual is not to be self-contained). Defer user procedures, formats, units, etc., to the user manual (Appendix F), unless user and operational guides are combined in one manual.

3.4 System Interfaces

Identify and describe interfaces between the operational environment and the hardware and software systems. Discuss data and control input media and devices, output devices, online and offline mass storage media, special devices, system software interfaces, system software services, etc. Identify protocols with systems-level personnel required for operations, such as private file assignments, assignment of job codes, billing, etc., but leave procedures for Section 5.1.1.

3.5 Data Base and Library Interfaces

3.5.1 Data Base Interfaces

Describe all data files in the data base that are referenced, supported, or kept current by the program, insofar as these are visible to the operator(s) of the program. Include the name and purpose of each such file, but defer detailed formats to an appropriate appendix (if these are necessary for operation). If there are offline or manually maintained parts of the data base that interface with program operations, describe these elements.

3.5.2 Library Interfaces

Describe any appropriate operator interactions or interfaces with document libraries, software (program, subprogram) libraries, or offline storage libraries. Reference source documents for data preparation and editing aids, output data monitors and diagnostic aids, etc., as applicable to operations.

4. OPERATIONAL FEATURES

This entire section documents the end-to-end operational characteristics of the program to a level of detail required for stand-alone usage of the manual. This section should describe each of the operator functions and options fully, giving examples of each, annotated and explained. Usage of graphic material in explanations is encouraged.

4.1 Operational Overview and Capabilities

Introduce the detailed operational functional characteristics to be discussed in the other subsections of this section by giving an overview of operator functions, operational concerns and activities, and program operational characteristics. Typical coverage at this point might address the structure, I/O and data flow, operational categories, modes of operations, security/protection measures, options, etc., as viewed by the operator(s). Use graphics to aid reader comprehension.

4.2 Inputs, Operations, and Outputs

Describe each of the features of the program, as visible to or of interest to, the operator(s); supply sufficient detail that the operator(s) may apply the procedures in Section 5. (Pertinent input, operations, and output characteristics will normally be integrated together in a narrative fashion, feature by feature. However, the outline below is segmented into three separate subsections so as to delineate specific items to be considered for discussion.)

4.2.1 Operator Input Characteristics

Define the requirements of receiving and processing user requests, input data, parameters, and controls. Typical considerations are (1) purpose or conditions, e.g., to make needed revisions to data base; (2) frequency, e.g., periodically, randomly, or as a function of an operational situation; (3) origin of request, e.g., network operations, program office, etc.; (4) medium, e.g., punched card, manual keyboard, magnetic tape; (5) restrictions, e.g., amount of data, priority, use authorization, security limitations; (6) quality control, e.g., need for checking reasonableness of input data, etc.; (7) disposition, e.g., requirements for retention, return, release, or distribution of input data received.

4.2.1.1 Input Format

Provide the layout forms and syntax of operator inputs as necessary. Include a description of each entry, with adequate grammatical rules and conventions used in each case. Distinguish literal input from syntactic variable identifiers. Typical considerations include (1) length, as characters/line or characters/item; (2) format, as, for example, left-justified free-form with spaces between items; (3) labels, tags, or identifiers; (4) sequence, or the order of placement of items in the input; (5) punctuation, or use of spacing and symbols to denote start and end of input, of lines, of data groups, of items, etc.; (6) rules governing the use of groups of particular characters or combinations of parameters in an input; (7) the vocabulary of allowable combinations or codes that must be used to identify or compose input items; (8) units and conversion factors; (9) optional elements and repeated elements; (10) controls, such as headers or trailers.

4.2.1.2 Sample Inputs

Provide specimens of each type of complete input form used by the operator. Such specimens should include, as applicable: (1) control or other header information denoting class or type, date and time origin, by the program; (3) trailer, denoting the end of input and other control data; (4) indication of omissions, i.e., classes or types of data that may be omitted, or are optional; (5) indication of repeated data, i.e., classes or types of data that may be repeated, and the extent of such repetition.

4.2.2 Operational Characteristics

Detail the operational characteristics of the program, and identify inputs, controls, and outputs associated with each. Typical coverage may address requirements to monitor indicators, devices, etc., and to interact with the program, special devices, other operators, users, *et al.*

4.2.3 Output Characteristics

Describe each of the output forms or other program responses to the operator in sufficient detail for his effective interpretation in the stated situation. Typical considerations include (1) use, e.g., by whom and for what purposes; (2) frequency, e.g., weekly, periodically, or on demand; (3) variations, e.g., modifications that may appear on the basic output; (4) destination, e.g., which users or work area; (5) medium, e.g., printout, punched cards, CRT display; (6) quality control, e.g., requirements for identification, checks for reasonableness, authorization to edit or correct errors; (7) distribution, e.g., requirements for retention or release, distribution, transmission, priority, security handling, and privacy considerations.

4.2.3.1 Output Formats

Provide a layout of each operator-pertinent output, with explanatory material keyed to the particular parts of the format illustrated. Include (1) header, e.g., title, identification, date, number of output parts, etc.; (2) body, e.g., information that appears in the body or text of the output, columnar headings in tabular displays, and record layouts in machine readable outputs, noting which items may be omitted or repeated; (3) units and conversion factors for numeric fields; (4) legends for abbreviated data; (5) accuracy; (6) trailer, e.g., summary totals, end-of-output labels, etc.

4.2.3.2 Sample Outputs

Provide illustrative examples of each type of operational output. In each case, discuss (1) the meaning and use of control data applied; (2) the source and characteristics of the data processed; (3) pertinent facts about the calculations made by the software; (4) characteristics, such as the presence or absence of items under certain other conditions of the output generation, other ranges of input values, or different units of measure.

4.3 Performance Characteristics

Describe the performance characteristics of interest to the operator, including where appropriate: (1) quantity of input and output; (2) throughput rate; (3) cost of service; (4) turn-around time; (5) reliability; (6) quality of service.

4.4 Security and Privacy

Describe security and privacy measures implemented in the program that restrict operations or guard data integrity via authorization keys, priorities, protocols, etc. Instruct the operator what features are operative within the several authorization levels, and identify penalties for inadvertent or malicious misuse. Provide warning and cautionary information, if applicable.

4.5 Operational Constraints

Describe specific restrictions that are placed on program features and options by the program (internal) design or by the operational or system environment (external). Give a brief statement of supporting rationale for such constraints and limitations, if deemed to be of value in operator understanding. Defer descriptions of procedural constraints to Section 5.5.

5. OPERATIONAL PROCEDURES

This section of the document should provide a concise and complete specification of the operational procedures to be followed by an operator for initializing, running, and terminating program operations in a correct and efficient manner. Additionally, material should be provided that will allow an operator to recognize alarms or improper operating conditions and to initiate recovery or reinitialization procedures with a minimum impact on overall program operations.

The material outlined here may be integrated as appropriate, rather than segmented as delineated in the hierarchy below.

5.1 Setup and Initiation

5.1.1 Protocols

Describe the operations protocols necessary to initiate a run, submit input, and receive output. Discuss, as appropriate: (1) opening a computer work order; (2) assignment and use of passwords and account codes; (3) authorization to use system and/or data base files; (4) assignment of permanent private files; (5) instructions for pickup or delivery of I/O interactive and batch protocols, etc.

5.1.2 Setup Procedures

Describe the setup procedures as a series of sequential steps or checklist that must be performed in preparing for operation. Identify all equipment (computer, standard peripherals, special or unique hardware, etc.) and software required for successful operation.

5.1.2.1 Hardware Setup

Provide complete instructions and procedures for connection (or disconnection) of hardware elements required for each given application. Describe the hardware configuration for each in terms of modes, operational controls, and data flow paths. Include annotated diagrammatic representations where appropriate. Differentiate between input and output devices. Specify any special calibration procedures necessary to verify the operational condition of individual equipment or of the total system.

5.1.2.2 Software Setup

Provide complete instructions for program setup, loading, and initiation of execution. Identify the software to be loaded and the medium on which it resides. Identify the peripheral from which the program is to be loaded. Describe in detail any manual fill procedures to be followed in loading the software into memory. List any loader prompting messages that might be directed to the operator. Indicate the operator inputs required to respond to prompting or to control the loading process. Identify loader responses to operator input. Provide examples of the form and content of operator inputs and system responses. Refer to appropriate appendices wherever applicable.

If different operating environment (system, equipment, or program) configurations are needed for different types of runs (such as pre-run checkout or calibration runs), then give setup and initialization procedures for each. Identify critical operations, alarms, and error messages.

5.1.3 Initialization

As in the setup procedure, describe the initialization process in terms of a series of sequential steps that must be performed to properly initialize the software for execution. Initialization should introduce the software to the environment, identifying the control source, the hardware/software configuration, I/O constraints, etc. If system peripherals such as disks or magnetic tapes must be initialized, indicate the procedures necessary to prepare them for program operation.

5.1.3.1 Parameters

Identify standard, nominal, or default values of parameters already incorporated into the program. Specify parameter values that the operator

is required to initialize and give instructions for doing so. If initialization can be performed from a remote source, reference the remote operations manual. Identify any constraints placed on parameter values with regard to any specific mode of operation, such as manual or automatic operation.

5.1.3.2 Calibration

Initialization may also require the calibrating of special equipment that can only be performed in conjunction with the operating software. Specify the procedures to be followed during such a calibration.

5.1.3.3 Operational Interaction During Initialization

List prompting messages that may be directed to the operator. Indicate the operator inputs required to respond to prompting or to control the initialization process. Identify the peripherals to which system output is directed and list the responses to operator inputs and program responses.

5.2 Input Handling

Describe the procedures for gathering input data and putting it in the format required for operating the program. Such procedures might include: (1) the method of extracting data from source documents or files; (2) usage of data preparation and editing aids or other software; (3) usage of special services, such as keypunch operators; (4) a checklist to determine rapidly if everything has been done; (5) special considerations for alternative input media; (6) special considerations for batch vs. interactive operation; (7) quality control, e.g., instructions for checking reasonableness of data, actions to be taken when data appears to be received in error, documentation of errors, etc.; (8) disposition, e.g., instructions for retention, return, release, or distribution of input data received; etc.

5.3 Operational Control Procedures

5.3.1 Operational Control

Operational control denotes program control after setup, initiation, and initialization have been completed and program execution has begun. During execution, especially in real-time operations, operator interaction with the system may no longer be confined to a series of sequential inputs, but, instead, may be described as a reaction to system behavior.

If processing requires or permits interaction or monitoring by the operator, provide instructions for terminal operations. Describe (1) data or parameter input devices and procedures; (2) control devices and instructions; (3) cassette/tape device operation; (4) run interruption/

recovery; (5) special terminal devices, e.g., plotters; etc. Provide samples to illustrate each.

5.3.2 Termination

Describe procedures for terminating operations as an orderly, sequential series of steps. Differentiate between normal termination and abnormal or emergency termination, and describe the conditions associated with each type. Describe briefly the effect of termination on I/O devices so that peripherals are not deallocated prematurely. If a summary of program activity is available, specify the procedure for generating such an activity report. If system prompting is provided, list the messages and identify the operator inputs required to respond to each. Refer to appropriate appendices if applicable. Give examples as appropriate.

5.4 Output Handling

Describe policies and procedures within the operational environment for handling the output data. Typical instructions cover (1) handling, disposing, disseminating, and routing of the various forms of output; (2) storing or archiving of output items for their later retrieval; (3) status reporting based on output parameters; (4) extracting and summarizing of information; and (5) checking, auditing, or inspecting the output data.

5.5 Operational Restrictions and Limitations

Identify and explain exceptions and restrictions in the procedures for receiving or preparing or entering input, operating the program, or handling the output. Such material might address (1) limited availability of input devices; (2) security considerations; (3) processing cost vs. time-of-day limitations; (4) restrictions on amount of input or output; etc. The restrictions and exceptions discussed here are restrictions in operational procedures, rather than in the program applications.

5.6 Interrupt, Recovery, and Restart Procedures

If not adequately covered in other parts of the manual, describe (1) detection criteria and procedures for anomalous program behavior; (2) meanings of error messages, codes, or indicators; (3) prescribed emergency actions by the operator; (4) procedures for correcting input errors; (5) procedures for restart/recovery; etc.

Describe the type and form of status and performance parameters displayed by the program. Identify the peripherals to which such information is directed. Include tables or charts that list and define status and performance output. Include examples of actual display output, if available.

Provide a convenient method, such as a decision table, for identifying single and multiple points of failure, and state what operator action should be performed with each type of failure. List the steps to be taken by the operator for each method of recovery and reinitialization. Provide a list of operator inputs and expected system responses. Include examples of each type of recovery and reinitialization procedure. Refer to appropriate appendices where applicable.

6. OPERATIONS SUPPORT

This section of the operator manual contains instructions for providing the support functions necessary to the day-to-day operation of the software, not covered by standard procedures or by environmental considerations previously described.

6.1 Operational Utilities

Describe operational utilities used for supporting program operations, and give procedures for using them (or references to such manuals, if general-purpose and readily available), as applicable. Such utilities might include: (1) file management and maintenance facilities; (2) data conversion software; (3) program test benchmarks, test aids, test generators, etc.; (4) hardware-calibration programs; (5) diagnostic or debug aids; (6) performance and utilization monitors; (7) data editors; etc.

6.2 Management Information Reporting

Describe requirements and procedures for management information reporting incurred by program operations. Topics in this section might include such things as (1) requirements for reporting, e.g., frequency, conditions, report type, etc.; (2) accounting procedures; (3) operations schedules; (4) resource management; (5) utilization statistics; (6) anomaly status; etc.

6.3 Logistics

Insofar as it is the responsibility of the readers of the manual to supply expendable materials for the operation of the program, provide procedures or references that describe how such materials are to be obtained, stocked, distributed, etc., unless these are items covered by operational standards cited earlier.

6.4 Anomaly Reporting

Describe requirements and procedures for submitting operational problem/failure reports to the proper program maintenance personnel, or for notifying management, users, or others of anomalies detected and repaired (as an operational service).

7. APPENDICES

Appended material may include, but are not limited to, explanatory material and references of an auxiliary nature, inserted directly or bound separately for convenience. The following suggested topics are typical. Appendices may be designated as "Appendix A," etc., if desired, rather than by the Dewey-decimal system given here.

7.1 Glossary

Give a list of mnemonics, acronyms, and unfamiliar or specially used terms appearing in the manual; provide definitions for each.

7.2 References

Provide a bibliography of references to other documents appearing in this manual. Give a brief indication or abstract of what is being referred to in each work.

7.3 Input and Output Formats

Provide detailed formats and syntax for operator inputs and outputs, together with associated response(s) as supplementary material to Section 4.2. Define, as appropriate: (1) data base I/O formats, parameters, and control characteristics; (2) communications device I/O formats, parameters, and control characteristics; etc.

7.4 Auxiliary Tables

Assemble in tabular form auxiliary reference material needed for program operations that is better located in an appendix rather than in the text proper. Display each table in a separate subsection (7.4.i) and introduce or explain the use of each table narratively.

7.5 Summary of Operations

Provide an abbreviated description of each of the operational features for the knowledgeable operator. This summary should be devoid of tutorial explanations, containing, instead, only technical descriptions or definitive examples for quick reference. Such material may be annotated to index the summary forms to pertinent sections of the manual containing detailed information.

7.6 Detailed Examples

Display the operation of the program via sample runs from beginning to end. Show all input, indicate all interactions in a timely sequence, and display all operator-pertinent output. Give examples of normal and abnormal runs, and illustrate the procedures followed in each case.

RECEDING PAGE BLANK NOT F

APPENDIX J

SOFTWARE TEST REPORT CONTENTS

This appendix contains an outline of topics typically considered for inclusion in the Software Test Report (STR). The items listed are not exhaustive nor are all those given necessarily applicable to a particular development. Rather, the topics contained herein should be considered as candidates for coverage in software test plans and test reporting.

The STR is a summary of test plans and test results; but it may, in some cases, also include the test archives (by attachment or separate volumes), if these are deemed to be of value for historical purposes or for reference in later maintenance activities. Insofar as test requirements are contained in the SRD, test plans are included in the SDD, and test specifications are part of the SSD, the STR need not repeat such material but merely reference it (when in compliance).

The STR contains the reviewable proof that delivery and acceptance criteria have been met, that the development phase can properly terminate, and that the program and its documentation are ready for a formal transfer into maintenance and operations activities. The outline below, therefore, covers the major items needed for technical and management review. The SRD may identify exactly what items of an optional nature should appear in the STR. However, at a minimum, the STR should be sufficient for verifying the quality, accuracy, and completeness of the software deliverables in meeting SRD requirements.

Figure J-1 is a top-level graphical outline of the document organization; greater detail is provided in the detailed outline that follows.

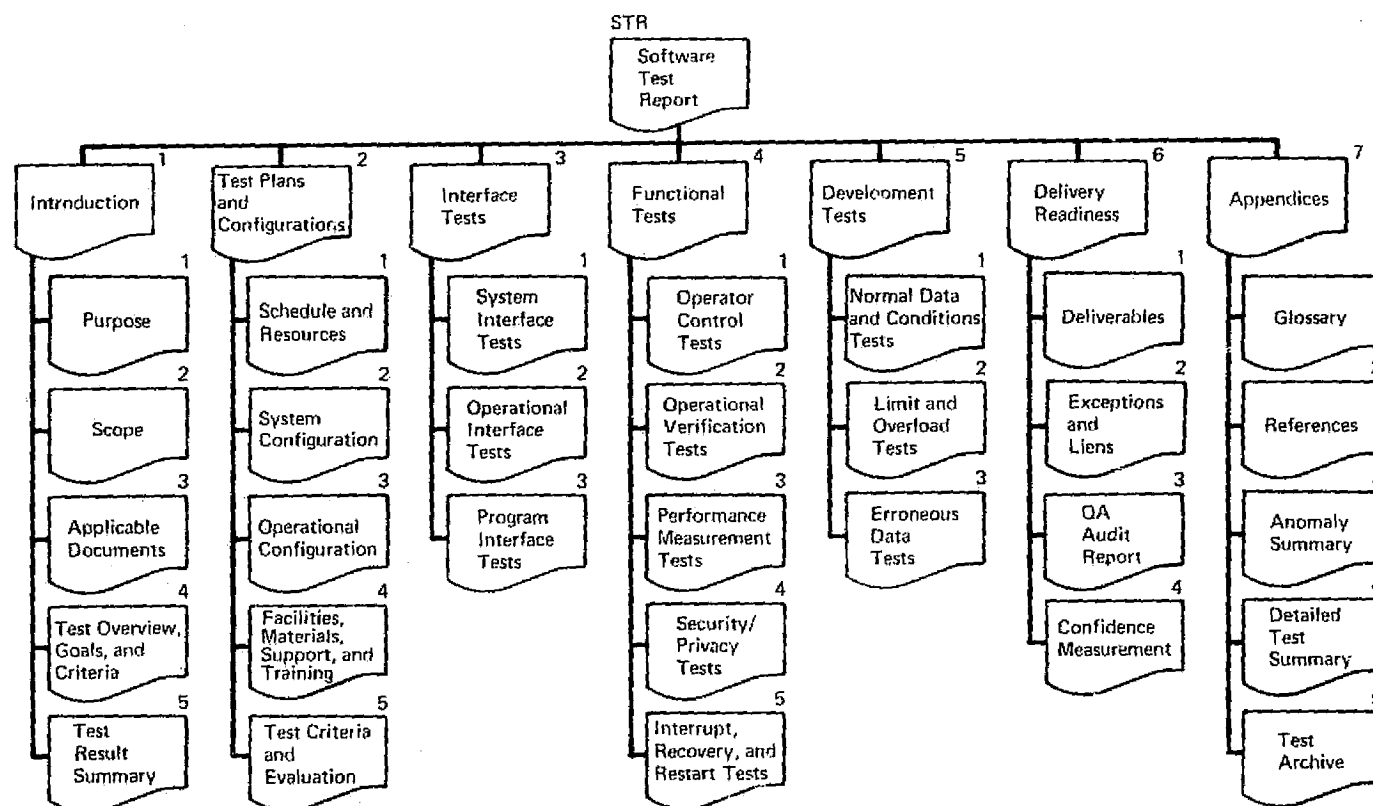


Figure J-1. Graphical outline of the STR

SOFTWARE TEST REPORT

Detailed Table of Contents

FRONT MATTER

Title Page. Provide a title page containing (1) document number; (2) SOFTWARE TEST REPORT; (3) program, subsystem, and system titles; (4) publication date; (5) signature block with name of preparer and others, as required. Signatures are to be supplied at STR completion. Date reflects time of last change to any item in the STR.

Abstract. Give a brief abstract that summarizes the program test and acceptance-readiness status.

Change Control Information. Provide a statement that relates the level of change control and authority exercised on this document, and procedures for update.

Distribution List. Provide a distribution list of all parties with need-to-know status of the software acceptance readiness who are to receive copies.

Distribution Information. Provide a statement that tells how additional copies may be obtained.

Table of Contents. Provide the detailed table of contents for the STR, which lists section number, title, and page of every item with a heading.

TEXT OF REPORT

1. INTRODUCTION

1.1 Purpose

Provide a brief statement of the purpose of this document.

1.2 Scope

Define the configuration(s) tested and the extent to which this document reports on test plans, test results, QA audit findings, exception identification, etc.

1.3 Applicable Documents

List all controlling documents, applicable standards, etc., governing the production of the results reported herein.

1.4 Test Overview, Goals, and Criteria

Briefly describe the objectives, general philosophy, approach, and overall strategy of test activities. Highlight the program functions, test plans and goals, operational interfaces, testing environment, acceptance criteria, and QA involvement. Correlate this material with requirements stated in the SRD. Identify differences between test and operations interfaces and environments that could affect program operation or assessment of its acceptability.

1.5 Test Result Summary

Discuss the various test runs, the extent of these tests, and the results of the testing activity. Identify major discrepant items or problems, and give a brief summary of the QA audit. Assess the effects that differences between the test environment and the actual operational environment may have had on the demonstration of capabilities. State the degree to which the results comply with SRD requirements. Summarize resource utilization and schedule performance, as appropriate to the reviewing authority.

2. TEST PLANS AND CONFIGURATIONS

2.1 Schedule and Resources

Show the acceptance test schedule with events, milestones, resource (manpower, computer, etc.) utilization, and time-phased loading of resources as actually incurred.

2.2 System Configuration

Identify the system configurations used in testing and QA. State differences between test and operational configurations, and assess the impact of these differences on software acceptability.

2.3 Operational Configuration

Identify the operational configurations used during testing and QA. State differences between test and operational configurations, and assess the impact of these differences on software acceptability.

2.4 Facilities, Materials, Support, and Training

Identify the locations and dates of tests, participating organizations, special test equipment or software, personnel skills (user, operator, developer, clerical), and materials required for test activities (e.g.,

documentation, loadable program, test data, sample output, expendable supplies, etc.). Discuss training of test personnel type and level, the use of multi-shift operation, etc., as appropriate to report pertinent support-level activities.

2.5 Test Criteria and Evaluation

Describe the rules used to evaluate test results, prioritize anomalies, register QA faults, etc. Reference or restate such criteria given in the SRD. Cite existing standard testing criteria and SSD test specifications, as appropriate. Identify standard tests and benchmarks. State and describe exceptions to test standards and standard tests. Describe how these criteria fulfill SRD requirements, if necessary.

Describe the techniques used for manipulating the test data (manual, automatic) into a form suitable for evaluation, or for comparison of actual results with predicted results.

3. INTERFACE TESTS

3.1 System Interface Tests

Describe the system interfaces tested (including I/O), the software functions that exercise these interfaces, the tests that verify these interfaces, and the progression of tests, one to the next, if pertinent. In top-down development, these interfaces are normally checked early, but informally, during developmental (correctness) testing. Citation of the results of developmental tests may be sufficient here, in some cases.

3.2 Operational Interface Tests

Describe the operational interfaces tested, the software and human functions that exercise these interfaces, the procedures and test data that verify these interfaces, and the manner in which progression was made from one test to another, if pertinent. Although these tests may have been simulated during development testing, it is unlikely that the actual operational environment was verified. These tests, therefore, will probably require separate demonstration during the acceptance test phase.

3.3 Program Interface Tests

Describe the program interfaces with libraries, data bases, and other programs, the tests that verify these interfaces, the results of such tests, etc. In top-down development, these interfaces are normally checked early in development testing; separate demonstration for acceptance may not be necessary in such cases.

4. FUNCTIONAL TESTS

4.1 Operator Control Tests

Describe acceptance tests executed to verify operator controls and responses, and the extent and results of these tests. Assess the degree of compliance with software requirements document and operations manual.

4.2 Operational Verification Tests

Describe acceptance tests executed to verify end-to-end functional capabilities, diagnostic features, and error responses of the program, and the extent and results of these tests.

4.3 Performance Measurement Tests

Identify the performance parameters calibrated or measured by tests, and compare the results with the required or specified values (SRD or SSD). Assess the degree to which performance meets program requirements and specifications.

4.4 Security/Privacy Tests

If there are requirements (SRD) or specifications (SSD) to demonstrate security/privacy features of the program, then describe the tests and test results. Assess the degree of compliance with requirements and specifications implied by this demonstration.

4.5 Interrupt, Recovery, and Restart Tests

Describe tests and test results that demonstrate capabilities for program interruption and recovery or restart, as required (SRD) or specified (SSD). Assess the degree of compliance with requirements and specifications.

5. DEVELOPMENT TESTS

5.1 Normal Data and Conditions Tests

Summarize the results, extent, and scope of development (production) tests performed to verify the processing algorithms, data structuring, and I/O under normal input data and normal operating conditions.

5.2 Limit and Overload Tests

Summarize the results, extent, and scope of development (production) tests performed to verify or calibrate the program response to limiting-case and out-of-range input data, real-time operational overloads, and beyond-normal conditions that reflect on the input dynamic range of the program.

5.3 Erroneous Data and Pathological Conditions Tests

Summarize the results, extent, and scope of development (production) tests performed to verify or calibrate the ability of the program to detect and recover from erroneous input, real-time operational failures, and abnormal operating conditions.

6. DELIVERY READINESS

6.1 Deliverables

Give an inventory list of deliverable software items that identifies for each: (1) item; (2) identification number; and (3) release date.

6.2 Exceptions and Liens

Describe deficiencies of the software as demonstrated by tests, inspections, QA audit, or other means, and prioritize these into categories, such as: (A) critical to successful program operation as required; (B) degrades performance or increases operational risk; (C) does not prevent software from operating and satisfying requirements, but is operationally undesirable; work-around necessary. In addition, indicate those discrepant items that can be negotiated for acceptability, those that can be accepted under lien for removal by a specified date after delivery, and those that cannot be accepted in software transferred to operational status.

6.3 Quality Assurance Audit Report

Summarize the QA activities in certifying the program for acceptance. Report the results of audits and reaudits of the SSD documentation against the code, as well as other materials audited. Identify deficient or marginal items (code, tests, and documentation).

6.4 Confidence Measurement

Discuss the method used to calibrate the acceptance tests and the level of confidence planned in calibration testing. Summarize the results of such tests, and state the computed level of confidence in the program and in the ability of acceptance tests to uncover anomalies, if present (e.g., see Chapter 9).

7. APPENDICES

Appendices may include, but are not limited to, auxiliary material inserted directly or bound separately for convenience. The following topics are typical. Appendices may be designated as "Appendix A," etc., if desired, rather than by the Dewey-decimal system given here.

7.1 Glossary

List the mnemonics, acronyms, and specially used terms appearing in this document, and give a short definition of each.

7.2 References

Provide a list of source documents, standards, procedures, manuals, etc., referenced in this document. Indicate the subject matter and purpose of each such item in the list.

7.3 Anomaly Summary

Provide a summary list of all discrepancies in the software package. For each, give the anomaly number, the program version, the reported date, a brief description, the priority category (e.g., see 6.2, above), the urgency for correction, parties responsible for correction, present status, the means of disposition, and other pertinent information. Provide, then, as appropriate, a detailed breakdown or backup description and history of each of the individual discrepancies (probably a compendium of anomaly report forms, status forms, statistics forms, etc.).

7.4 Detailed Test Summary

When appropriate to include summary results of individual tests or more detailed descriptions of the tests performed, do so in this appendix. Discuss the general method or strategy of the testing, and describe: (1) the type, volume, frequency of input used; (2) the extent of testing performed (total or partial) with rationale for choice; (3) method of recording results and other information about the tests; (4) limitations on tests due to conditions, such as interfaces, equipment, personnel, and data bases; (5) controls, such as manual, semi-automatic, or automatic insertion of inputs, etc.; (6) sequence of operations or progression from one test to another so that the entire test cycle was complete; etc.

7.5 Test Archive

This appendix, if present, may contain the entire collection of test outputs over the entire project lifetime (probably as separate volumes), or may contain only material selected as supportive to this document or as necessary for future historical or reference purposes.

APPENDIX K

SOFTWARE MAINTENANCE MANUAL CONTENTS

The purpose of the Software Maintenance Manual is to provide the sustaining and maintenance personnel with information necessary to understand a program in its operating environment, and to list the procedures for correcting errors, updating documentation, making program modifications, etc. This monograph has always assumed that the SSD and user/operator manuals form the primary sustaining and maintenance source documents.

The maintenance manual, therefore, does not repeat this information. Instead, it gives procedural instructions for performing the sustaining and maintenance activities. As with others manuals described in these appendices, maintenance procedures should be accumulated and written concurrently with the other program production activities. Even when the maintenance and operations organization has different configurations and standards than implementors (and will, therefore, write this manual themselves), the skeleton of procedures used during development can be of tremendous benefit as the basis of the manual to be written. Figure K-1 is a top-level view of the suggested organization for a Maintenance Manual. The remainder of this appendix is a detailed outline with guidelines for writing the material suggested. As with the user and operator manuals, topics described in this appendix are presented in checklist hierarchy; actual writing of the manual may well integrate maintenance functions with interfaces and procedures, so as to form a set of monolithic descriptions of the subjects covered.

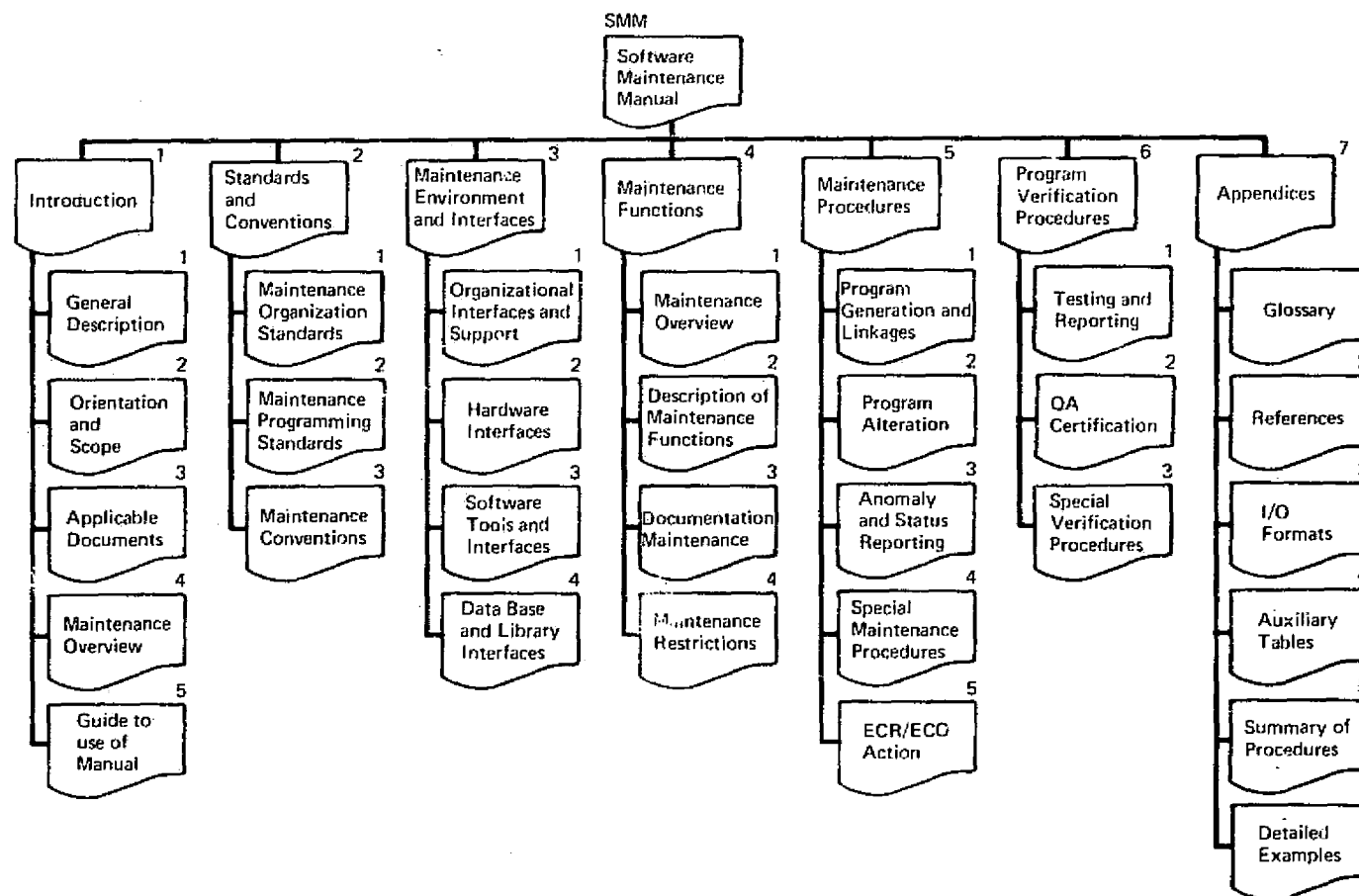


Figure K-1. Top-level view of the SMM table of contents

SOFTWARE MAINTENANCE MANUAL

Detailed Table of Contents

FRONT MATTER

Title Page. Provide a title page containing: (1) document number; (2) SOFTWARE MAINTENANCE MANUAL; (3) program, subsystem, and system titles; (4) publication date; (5) author and authorization signatures, as appropriate; and (6) releasing organization.

Abstract. Give a brief abstract that summarizes the purpose and usage of the manual.

Change Control Information. Provide a statement that specifies the level of change-control authority for the manual, and state procedures for submitting change requests and reporting of errors.

Distribution Information. Provide information that tells how copies of this document may be obtained.

Table of Contents. Provide a detailed table of contents for the manual, listing section number, title, and page of every item with a heading.

TEXT OF MANUAL

1. INTRODUCTION

1.1 General Description

Provide a brief statement that identifies the program, system, and environment covered by this manual. Give the program title, tag or label, and programming language(s). Perhaps also appropriate in this description are relationships to other maintenance activities, alternate environments, organizational responsibilities, and maintenance programmer responsibilities.

1.2 Orientation and Scope

Identify the intended users of this manual, their assumed backgrounds, the level of skill required, and the extent to which this manual is self-contained. Describe the scope of this manual as it pertains to the

completeness with which it covers maintenance matters. Identify any significant limitations in the program maintenance as levied by this manual.

1.3 Applicable Documents

Identify all documents, controlling or informational, which apply to or regulate the maintenance activities associated with the program.

1.4 Maintenance Overview

Provide a brief overview of the maintenance functions and the general policies and procedures that apply to each. Identify salient characteristics of the maintenance activities, such as: (1) change control; (2) resource estimation; (3) reporting; (4) operations/user liaison; (5) developer liaison; (6) division of labor; (7) support requirements; etc.

1.5 Guide to the Use of the Manual

Explain how this guide is to be used by maintenance personnel.

2. STANDARDS AND CONVENTIONS

This section describes standards imposed on the maintenance organization (or used by them) and conventions (e.g., notations and terminology) applied in this manual.

2.1 Maintenance Organization Standards

Identify or reference applicable existing organizational maintenance standards, state any exceptions to these standards necessitated by this program, and describe the alternate practices to be used.

2.2 Maintenance Programming Standards

Identify or reference applicable existing programming documentation and testing standards to be applied in program maintenance, state any exceptions to these standards necessitated by this program, and describe the alternate practices to be used. State the language(s) to be used for program modifications.

2.3 Maintenance Conventions

Define notations, terms, and other conventions or assumptions used generally throughout the manual. Include such items as ways of distinguishing literal fields from syntactic variables in descriptions of input and output formats, means for differentiating inputs and outputs in interactive examples of procedures, non-standard mathematical usage, special acronyms, etc. Describe maintenance conventions, such as file naming of scratch, update, and test files.

3. MAINTENANCE ENVIRONMENT AND INTERFACES

3.1 Organizational Interfaces and Support

Describe the maintenance organizational environment, responsibilities as appropriate to perform maintenance, liaison activities with others, support required for or available to maintenance activities, and other matters of an organizational nature, as beneficial to the readers of this manual.

3.2 Hardware Interfaces

Identify (by reference to system documentation if possible) the equipment required for operation and maintenance of the program. Describe any unusual features used. Include such information as (1) size of processor and internal storage; (2) online and offline storage media and devices; (3) online and offline I/O devices; (4) data transmission devices.

3.3 Software Tools and Interfaces

Identify the system support software needed for program maintenance, and describe the interfaces in enough detail so as to prepare the reader for the functions and procedures in sections to come. Include, as needed: (1) operating system; (2) compiler(s) or assembler(s); (3) debug facilities; (4) test data generators; (5) text editors; (6) data management systems; (7) report generators; etc. Include the version or release number of each and any unusual features used.

3.4 Data Base and Library Interfaces

3.4.1 Data Base Interfaces

Describe or reference documentation on the data bases required or typically used for maintenance, if any. Include, as appropriate, information such as keys, units of measurement, format, range of values, names of files, location of the data element dictionary, etc.

3.4.2 Library Interfaces

Identify programs, subprograms, or other materials in libraries that are used in program maintenance, and give appropriate references to detailed usage information. Give a brief description of the usage and interfaces pertinent to this manual.

4. MAINTENANCE FUNCTIONS

This section describes functions performed in maintaining the software, not the procedures for performing these functions.

412 Appendix K

4.1 Maintenance Overview

Describe the maintenance tasks in summary fashion as an introduction to the remainder of Section 4. If already covered in Section 1.4, so state, and omit duplication. Identify such functions as (1) anomaly detection; (2) anomaly correction; (3) anomaly report custodianship activities; (4) software modification activities; (5) engineering change control activities; (6) liaison activities; (7) operations support; (8) user support; (9) special maintenance functions.

4.2 Description of Maintenance Functions

Detail each of the maintenance functions introduced in Section 4.1, above, as separate subsections (4.2.i). Use a level of detail sufficient to describe the functions to the degree required to apply the procedures in Section 5.

4.3 Documentation Maintenance

Identify the program documents in the custody of maintenance, and describe the role and responsibility of maintenance personnel for keeping these documents current. Identify resource availability or limitations for such maintenance, if appropriate. List (or reference) criteria and approvals for making documentation modifications, quality requirements for redocumented items, requirements for QA audit or recertification, etc.

4.4 Maintenance Restrictions

Identify and describe or reference restrictions placed on maintenance functions and activities by the host system by supporting services, by change control requirements, by the maintenance organization, and by available resources.

5. MAINTENANCE PROCEDURES

This section describes, in robotic sequence detail, the steps that are used to perform the functions described in Section 4.

5.1 Program Generation and Linkage-Edit Procedures

Identify and describe each of the symbolic, relocatable, and absolute elements required to generate the program. Detail the method and steps used to install the program into its operational environment (ready for operations and use). Describe job control code, linkage-edit (map) code, etc., and the sequencing required. Cite references to appropriate manuals, detailing the various program generation tools used. Give specific end-to-end running examples of each case (e.g., backup tape, symbolic system files, relocatable system files, absolute elements, etc.).

5.2 Program Alteration Procedures

Detail the method, steps, and protocols used to correct anomalies or make other changes in the program code. Describe the use of text editors, assembler(s) or compiler(s), debug and other checkout facilities, binary or symbolic patch capabilities, etc. Give references to manuals describing the details of the software tools used. Give specific end-to-end running examples of each alteration method, complete with all of the job control code needed for execution.

5.3 Anomaly and Status Reporting Procedures

State procedures for handling anomaly reports, such as logging, regular status report generation, user/operator notification, determination of priority and method for removal, etc.

5.4 Special Maintenance Procedures

Describe any special maintenance procedures, such as scheduling of activities, redocumentation procedures, program backup (master) handling, security/integrity monitoring, data base recovery, and hints for using SSD and other manuals in maintenance activities.

5.5 Engineering Change Request/Order Action

Give procedures for instituting or evaluating change requests, for responding to change orders upon receipt, and for notifying upon compliance.

6. PROGRAM VERIFICATION PROCEDURES

6.1 Testing and Reporting Procedures

Describe the procedures for setting up tests and recertification of the software, either general or following modifications. Include references to test standards, test specifications, and benchmark test data, as appropriate. Give specific end-to-end running examples of such testing, including all of the job control code required.

6.2 QA Certification Procedures

Present the procedures to be followed in obtaining QA recertification of all changes, as required under change control restrictions.

6.3 Special Verification Procedures

Identify and give procedures for special activities associated with recertification of the software, such as operational demonstration, approvals, sending of materials to the program library, special system processor registration protocols, etc.

7. APPENDICES

Appendices may include, but are not limited to, explanatory material and procedures of an auxiliary nature, inserted directly or bound separately for convenience. The following topics are typical. Appendices may be designated as "Appendix A," etc., if desired, rather than by the Dewey-decimal system given here.

7.1 Glossary

List the acronyms, mnemonics, abbreviations, and specially used terms that appear in this manual, and give a short definition of each.

7.2 References

Provide a list of all source documents, standards, procedures, and reference material used for program maintenance. Indicate the subject matter and purpose of each reference.

7.3 Input/Output Formats

Provide detailed formats and syntax for input and output data used specifically in software maintenance. Define, as appropriate: (1) data base I/O formats, parameters, and control characteristics; (2) communications device I/O formats, parameters, and control characteristics; etc.

7.4 Auxiliary Tables

Assemble in tabular form all auxiliary reference data needed for software maintenance, which are better located in an appendix rather than in the text proper. Display each as a separate subsection (7.4.i), and introduce or explain the use of each table narratively.

7.5 Summary of Procedures

Provide an abbreviated set of procedures for each of the maintenance functions suitable for use by the knowledgeable user. This summary should be devoid of lengthy tutorial explanations, containing, instead, only technical descriptions or definitive examples for quick reference.

7.6 Detailed Examples

Display the application of maintenance procedures via samples from beginning to end. Show all input, indicate all interactions in timely sequence, and display all responses and output.

APPENDIX L

SAMPLE PROGRAMS FOR PROJECT MANAGEMENT

My original intent for this appendix, while I was writing Part I of this work, was to present end-to-end detailed program examples in which the standards of this text would be applied. Such programs could then be used as examples of Software Specification Documents illustrative of Class A and B levels of detail. I had hoped to enter more than the four examples you see here, but a compromise was necessary for reasons of space. The programs I envisioned were principally management-oriented, such as Examples L-1 and L-4; but design aids, such as Example L-2, and useful subroutines, such as Example L-3, were not to be left out altogether.

The programs of this appendix are rather small and based on well-known algorithms. Neither are they grandiose in range of application, but merely scratch the surface of project management and development needs. More extensive requirements for software management information systems appear in [47]. Useful programs for grading project performance, status reporting, data sorting, and linear programming appear in [48].

Small programs tend to have limited utility, unless they can easily be generalized to wider applications than usually dictated by their size. I certainly hope, therefore, that the standardized developments presented herein are both sufficient, structured, modular, and understandable as to permit such extensions to be easily forthcoming. However, the main benefit of this appendix more probably lies in using the given examples as models for documenting programming specifications.

Each of the examples follows the SSD outline given in Appendix E; however, only Example L-1 goes through a section-by-section consideration of topics. In each case, headings that did not apply were omitted; in some cases, where a theoretical background needed to be established for readability, other headings were invented.

Because the program descriptions are not directly or ultimately coupled to an unambiguous computer language, I realize that some readers may possibly interpret what they read here slightly differently than do others. Nevertheless, I have tried to make the designs as free of language or machine characteristics as I consciously could, and I hope the intent, if not the detail, is clear in each.

The first example (L-1) is what I consider to be formal Class A detail with respect to the programming specifications and fairly strict adherence to Appendix E, within the scope of the host-independence sought for. Some areas are not quite Class A in detail and need to be worked on before one could term the whole SSD as "Class A." In this sense, the SSD is seen in an embryonic stage of development, before much of the detailed test specifications, etc., have been written. The level presented may seem "overkill" for such a short program. For a large program, however, such detail may be essential for understanding.

Example L-2 is a much shorter program, but the detail level in the programming specification is still Class A, within host-dependent considerations. Standards, system environment and interfaces, test and verification details, and other lower-level considerations are not covered, so the document may fall into the Class B category, in that "qualified personnel (engineer or equivalent) using documented techniques and approved programming practices may be required to satisfactorily produce that item entirely from information supplied."

Example L-3 is even shorter, yet Class A detail prevails in the programming specifications.

The final example, L-4, is Class B; the level of detail in the programming specification section relies on the programmer's ability to design the code effectively, with discretion permitted if there is satisfaction of the program requirements with respect to performance and quality without unreasonable risk.

EXAMPLE L-1

GENERATING SCHEDULES USING THE CRITICAL PATH METHOD

1. INTRODUCTION

1.1 Purpose and Scope

This specification covers the design of a program to accept items from a project Work Breakdown Structure (WBS) to compute schedule information, such as critical path status, earliest starting time, latest start time, earliest finish time, latest finish time, and float (or slack) time for each task, and to print a schedule based on these computations.

The program, in the form specified here, does not recognize precedence of tasks based on resource availability, nor does it allow explicit usage of lag times, starting dates, or ending dates to be input by the user, as some PERT/CPM systems do. Lag times may be simulated, however, as separate tasks appropriately inserted into the network.

1.2 General Description

The basic critical-path method of schedule network analysis employed here is discussed in several sources; References 7.2.1 and 7.2.2 are recommended to the interested reader. Extensions to the basic method appear in many PERT-like systems, such as the IBM Project Control System (Reference 7.2.3).

The capability described herein assumes that work task identification, task duration, and task precedence-successor relationships are input from an unspecified medium into memory for access by the computation modules. Results of computations are printed on a line-oriented, paged-output device.

Further details regarding the function of the program are contained in Section 4, and programming specifications appear in Section 5.

2. STANDARDS AND CONVENTIONS

2.1 Specification Standards and Conventions

Specifications contained herein shall, on completion of this document, reflect A-1 documentation quality, as defined in Reference 7.2.4. Flowcharts

and accompanying narratives are used to define the programming specifications (Section 5); these are governed by Reference 7.2.5.

2.2 Programming Standards

Programming standards as specified in Reference 7.2.6 shall apply to all implementations except as otherwise dispensed by waivers stated herein.

2.3 Test and Verification Standards

Tests and verification activities are governed by Reference 7.2.7.

2.4 Quality Assurance Standards

QA standards as defined by Reference 7.2.8 and further refined in Section 6.3 shall apply as condition for delivery.

3. ENVIRONMENT AND INTERFACES

This specification is meant to be system-independent in the sense that it places few restrictions on peripheral devices and memory, except that (1) an input medium must be available to input the WBS network; (2) sufficient storage is available for holding the network and the program; and (3) an output device is available for displaying the schedule information. If insufficient fast storage is available for the network, but random-access mass storage is available, then suitable alterations in the program can be made to accommodate such a configuration.

The coding language is also unspecified; however, the procedural descriptions used presume that strings of characters can be input and output, and that such strings can be specified by name, or by name and index value, if arrayed. Coding in lower-level languages not possessing string operations must simulate these features as subroutines or functions.

4. SOFTWARE FUNCTIONAL SPECIFICATION

4.1 Functional Organization and Overview

A task in a Work Breakdown Structure, as assumed by this program, is specified by (1) a task code, (2) a task description, (3) a duration, and (4) a list of codes of tasks that must terminate before the current task may initiate (for reasons decided by the user). No task may initiate until all of its named preceding tasks have terminated. Special tasks with zero duration may be inserted (these are called "milestones") so as to enable the definition of schedule networks in which tasks may start synchronously (a start-to-start relationship) or terminate synchronously at a milestone (an end-to-end relationship), in addition to the normal precedences (end-to-start relationships).

The *earliest start* of a task is defined as the maximum earliest finish of all the listed required prior tasks; the *earliest finish* of a task is defined as its earliest start plus its duration. The *latest finish* of a task is defined as the minimum latest start of all listed required subsequent tasks; the *latest start* is the latest finish less the task duration.

One special milestone (zero-duration task) must be identified as the "starting" task, having no list of preceding task codes; this task also has a "starting date" defined. A "project termination" milestone is optional, supplied by the program, if omitted; the earliest start and latest finish of this task are set equal.

The *float*, or *slack time*, of a task is defined as the difference between its latest and earliest start. A task with zero float is said to be a *critical task*. There always exists at least one set of critical tasks that span, end-to-end, the entire project; these tasks are said to form a *critical path* in the project workload. Critical path tasks are indicated on the printout by an asterisk (*).

If a critical-path task slips in schedule, the project termination date slips an equal amount. Any non-critical task may slip by an amount up to its float without causing end-date slippage.

The input unit of all durations is *whole days*. Output schedules then account for normal work week and holidays by way of a calendar data base residing in an unspecified sequential file medium (e.g., disk or cards). The output schedule consists of a list of tasks sorted in topological order, giving early and late start and finish dates, plus the float time. In addition, early and late start and finish days of work since project start are displayed on the printout, with critical tasks identified.

4.1.1 Detection of and Recovery From System Failure

This specification does not address system-dependent features of the host environment.

4.1.2 Detection of and Response to Data Input Errors

This specification does not cover syntactic checking of data elements read from the work breakdown structure or calendar file. Such checking, however, shall be incorporated into the input functions of the program that read the pertinent data described herein. Such functions shall emit diagnostic error messages that describe the syntactic violation, and shall then operate as if an end-of-file had appeared.

4.2 Configurations and Modes

The program described herein has only a single mode of operation, without options. Consult Section 7.3 for suggested future options.

Input, Processing, Output Specifications

Many of the formats for input data, as well as other matters, are left somewhat open, to permit flexibility in implementations on host systems with differing characteristics. Such matters may be resolved arbitrarily, so long as the results do not conflict with other specifications contained herein.

Inputs to the program are as follows:

- a. For the project start (the first-received task, with no predecessors):

milestone code	string of characters
milestone title	string of characters
start date	day of year

The input format, date format, and string lengths are not covered by this specification. The output format specified in 7.5 provides space for displaying up to 10 and 32 characters, respectively.

- b. For each task, from the same medium as (a), above:

task code	string of characters
task title	string of characters
duration, in days	integer
list of precedent- task codes	task codes, above

The input medium, mode of entry, and format of these items are not covered by this SSD. Milestones (except start) are input as tasks with zero duration.

- c. From the calendar file, for each day in sequence, extending over the entire project lifetime:

workday	boolean
date	month, day, year

The actual detailed configuration and format of these data are not covered by this specification.

Although precise input details have been excluded from the above items, the programming specifications of Section 5 do show definitive requirements for order of input and data type.

Two task codes are reserved. These are:

END
FINISH

The END task code is a signal to the program that there are no further tasks; the network data is complete. The END record is, therefore, the last in the WBS file. An attempted read resulting in endfile shall return "END" as the item.

The FINISH task code is used internally within the program to name the project termination milestone. No title is supplied for this milestone by the program. The user may use FINISH as a task code and supply a title; however, any task naming FINISH as a predecessor will be flagged as an error. Successor linkages to FINISH will be erased by the program.

Processing specifications for the program are as follows:

- a. Accept start milestone and task descriptions, as above. The first task encountered is assumed to be the start.
- b. Topologically sort the tasks (Reference 7.2.9) into a list in which all precedent tasks for a given task precede that task in the list.
- c. Scan this list (it begins with the start milestone), computing start and finish times (defined in Section 4.1, above) for each. Identify critical tasks.
- d. Detect and display errors in improperly formatted input, insufficient calendar input, and circular or disconnected networks.
- e. Print the schedule.

Output from the program consists of the schedule report and error diagnostics. The output schedule takes the form of a report which lists the tasks in topologically sorted order, showing early/late start/finish dates, float values, and times past project start. The output format is shown in Figure 7.5.1.

5. PROGRAMMING SPECIFICATIONS

5.0 Program Overview

The SCHEDULER program consists of four subprograms that execute in sequence:

BUILD NETWORK	(BUILD)
TOPOLOGICAL SORT	(TOPSORT)
CALCULATE DATES	(DATES)
DISPLAY SCHEDULE	(DISPLAY)

The representation of the schedule network within the design is the chief item required for understanding the procedures in the remainder of this section. There is a certain set of information that is input or calculated for each task, represented by a node, in the network:

information	identifier	type
TASK CODE	CODE	string
TASK TITLE	TITLE	string
TASK DURATION	DUR	integer
EARLIEST START	EST	integer
EARLIEST FINISH	EFIN	integer
LATEST START	LST	integer
LATEST FINISH	LFIN	integer
FLOAT TIME	FLOAT	integer
NUMBER OF PREDECESSORS	COUNT	integer
POINTER TO LIST OF SUCCESSORS	TOP	"pointer"

The first three items are input directly, the next six are calculated after sorting, and the final is supplied as a result of building the network. The node data structure and network relationships are organized into a simulated list using arrays named with the identifiers listed above, along with the two arrays:

information	identifier	type
SUCCESSOR NODE	SUC	"node"
NEXT LIST ITEM	NEXT	"pointer"

The "pointer" data type in the two tables above is an integer index into the SUC:NEXT arrays; the "node" data type is an integer index into the CODE:TITLE:DUR:...:TOP arrays. In this way, information about a node (task) is locatable via the node index, and its successor nodes can be found following the node TOP pointer to SUC, for the first, and thence via NEXT pointers to the remainder. This representation is discussed further in Knuth (Reference 7.2.9). The apparent node representation is shown in Figure 5.0.1; the network is implicit, then, in COUNT and SUC relationships.

Languages that have "ARRAY OF RECORD" syntax may code each node as an indexed record.

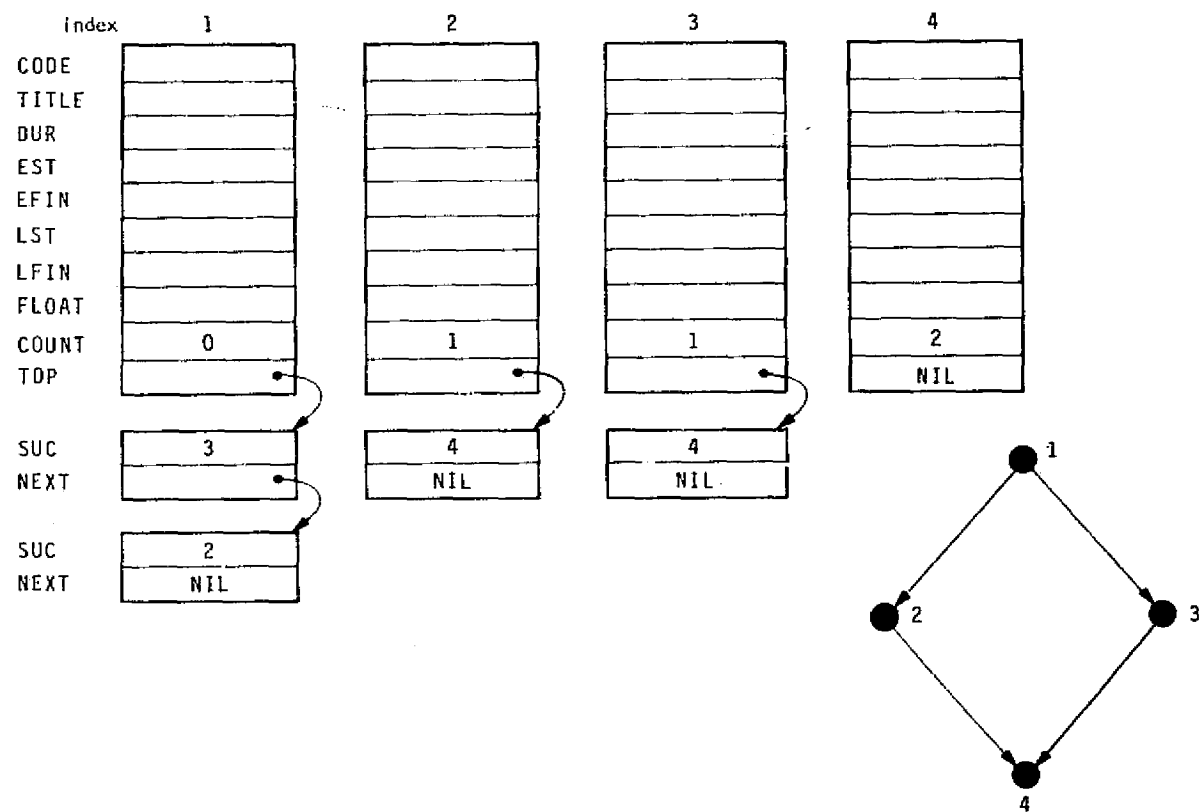


Figure 5.0.1. A simple precedence graph and its structural representation

5.0.1 Building and Sorting the Network

The process of building the network consists of reading the WBS file and storing task information in the node arrays. As tasks are read, each task index is recorded in an integer topological-sort list array, TSORT. Indexes of nodes having a zero predecessor COUNT are queued into TSORT from the front; these are already in sorted order. The others are queued at the rear of TSORT. Such a list permits the processing of tasks independently of the way the search algorithm has entered the nodes into memory. Thereafter, the topological sort procedure considers, in turn, each node in the front segment of TSORT, "removing its edges" in the network by reducing the COUNT of nodes identified as successors. When a COUNT field of a node hits zero, that node index is inserted at the rear of the front segment of TSORT. If some nodes still have non-zero COUNTS after all of the front segment of TSORT has been processed, a loop in the network exists (identified as an error).

5.0.2 Calculating Schedule Times

Early start and finish times are calculated by scanning the nodes listed in TSORT in forward order; then late start, late finish, and float values, by scanning TSORT in reverse order.

Dates are assigned to the project times by way of the CALDR array, filled from the calendar file. The CALDR array is an ordered set of strings indexed by work day. More precisely, when filled, the value of CALDR(0) is the starting date (month/day/year) corresponding to START, read in from the project start milestone. START is an integer that defines the day of year for beginning the calendar file read-in. The size of the CALDR array permits entries indexed 0 through MAXDATE. If the project goes beyond MAXDATE days, only the first MAXDATE days appear in the array. If the calendar file does not extend far enough into the future to provide dates for the project times, then "DAY *n*", where *n* is the project work day, is entered into the CALDR array.

Further details are contained in the programming specifications in the remainder of this section.

5.0.3 Program Tier Chart

The list below presents the modular nesting of program elements:

```

1  SCHEDULER
    .1  INITIALIZE
        .3  HEADER_DATA
    .2  BUILD
        .6  REGISTER
            .1/S1  SEARCH
            .6  CONNECT
                .3/S1  SEARCH
                .7/E1  ERROR
        .8  TERMINATOR
            .1/S1  SEARCH
            .3  CHECK_SUCCESSORS
                .3/E1  ERROR
            .6  LINK_TO_FINISH
                .4/E1  ERROR
    .5  TOPOSORT
        .4  ERASE_EDGES
        .7  DIAGNOSE
            .1/E1  ERROR
    .7  DATES
        .1  EARLY_DATES
            .4  SUCCESSOR_DATES
        .3  LATE_AND_FLOAT
            .4  TASK_LATE_DATE
    .8  DISPLAY
        .1  CALENDAR
            .1, 2, 10/E1  ERROR
            .6/F1  STR
E1  ERROR
F1  STR (Integer-to-string conversion)
S1  SEARCH
    10/E1  ERROR
  
```

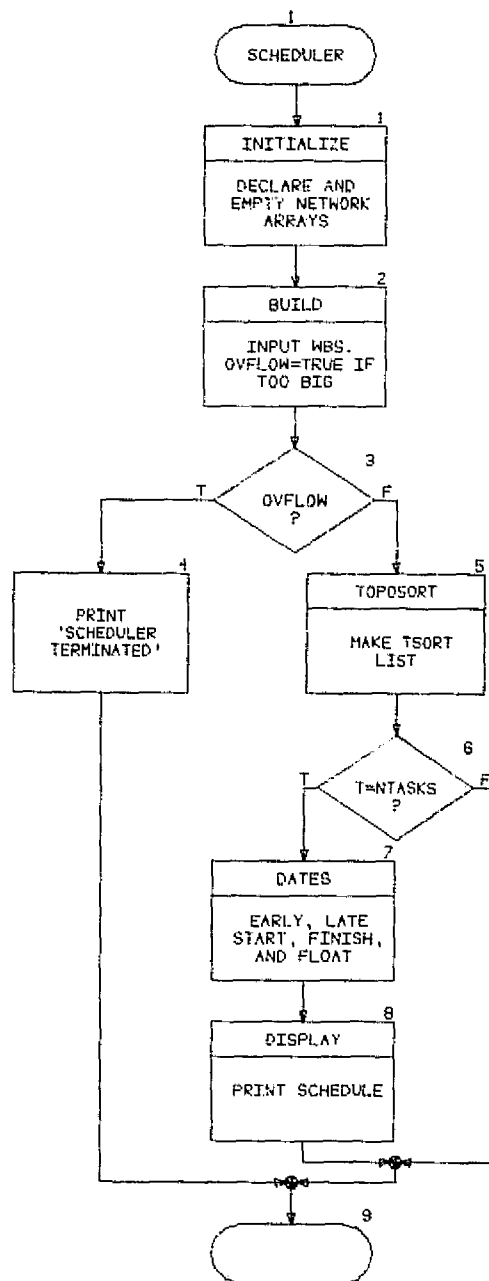
Indentation in this list denotes procedural nesting of the modules named. Numbers are Dewey-decimal module codes (e.g., TERMINATOR is found as module 1.2.8).

Chart Number	1
Module Name	SCHEDULER
Date	4/1/77

5.(1) Critical-path SCHEDULER program

- .1 Declare global schedule network arrays and set all numeric items to zero, all string items to null. Prepare the HEADER data for the report in step 8.
- .2 In building the network, if either the task node arrays or the successor linkage arrays become filled prematurely, print an error message. Add a project termination milestone to the schedule network. Return a flag OVFLOW with false value if the WBS input did not exhaust the network arrays; true, otherwise. NTASKS records the number of tasks entered.
- .3 A true value of OVFLOW terminates the scheduler,
- .4 printing a message before the program terminates.
- .5 If the network was input without overflow, then the TSORT list contains the topological sort. If the network contains a loop, print "WBS IS CIRCULAR AMONG ITEMS" and give the list of task codes. Identify one such loop. T records the number of sorted items entered into TSORT.
- .6 If all tasks are not in the list, the WBS is circular, so terminate.
- .7 Otherwise, scan the list forward for early times (see definitions in Section 4.1), and in reverse for late times (Section 4.1). Calculate float times during the second scan as well.
- .8 See Sections 4.3 and 7.5 for details of output format and content
- .9 Perform any cleanup necessary in the coding language (e.g., closing files) before program termination.

Example L-1 427



SCHEDULER
1 APR 77

D:	1/12/78
C:	1/19/78
A:	1/19/78

1 JAN 78

Chart Number	1.1
Module Name	INITIALIZE
Date	4/4/77

5.(1.1) INITIALIZE the scheduler program

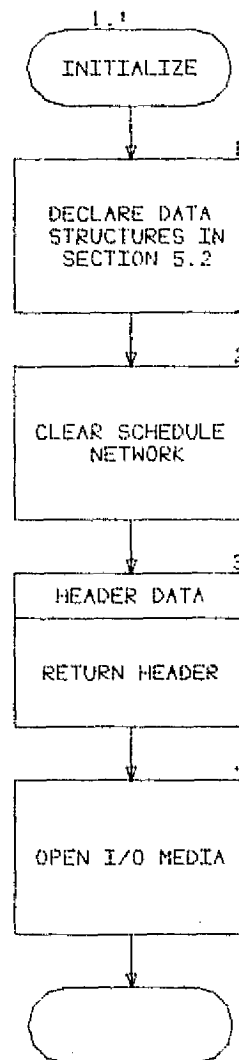
On entry, no data declarations have been made.

This procedure declares all data for the entire program, clears the schedule network by setting all numeric values to zero, all strings to null, and stores the information necessary to generate the top of the schedule report in HEADER for later access by DISPLAY/1.8.

On exit, the program is ready for execution.

- .1 All data declared as specified here is assumed globally available to all modules in this program. These are placed all together so as to be statically allocated at compile time, should the coding language demand it; these are placed first, in case declarations are executable. In assembly language, it may be necessary or desirable to locate the code for such declarations elsewhere.
- .2 Strictly speaking, only EST, COUNT, and TOP must be cleared; CODE, TITLE, and DUR are later input, EFIN, EST, LST, LFIN, and FLOAT are computed. SUC and NEXT need not be cleared, either; however, later modifications may change some of these assumptions. Therefore, clear them all.
- .3 HEADER information prepared at this point is used by the DISPLAY step 1.8.7 to achieve the format in 7.5.
- .4 Initialize I/O media as required to input WBS and calendar data in later parts of the program.

Example L-1 429



1.1
INITIALIZE
4 APR 77

D:	pen	1/18/78
C:	KC2	1/18/78
A:	pen	1/19/78

11 JAN 78

Chart Number	1.1.3
Module Name	HEADER_DATA
Date	4/4/77

5.(1.1.3) Generate HEADER DATA

The module described here at present is a STUB.

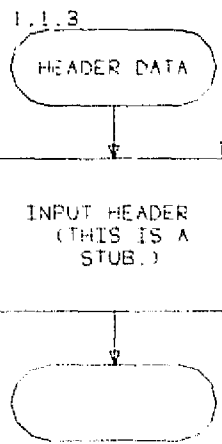
On entry, data structures have been declared.

This procedure obtains information from an input medium for use in printing the output header.

On exit, the string HEADER contains the title to be used in the first line of the schedule report (see 7.5).

Rationale: Decisions regarding exact input format have not been made. These decisions will be influenced by choice of programming language and possible interactive/batch operating modes. These decisions affect only the DISPLAY step 1.8.7, and may, therefore, be postponed until the other algorithms herein are verified.

Example L-1 431



1.1.3
HEADER DATA
4 APR 77

D:	LC2	11/13/78
C:	LC2	1/18/79
A:	leg	1/19/79

11 JAN 78

Chart Number	1.2
Module Name	BUILD
Date	4/4/77

5.(1.2) Procedure to BUILD schedule network

On entry, the schedule network has been declared, but cleared.

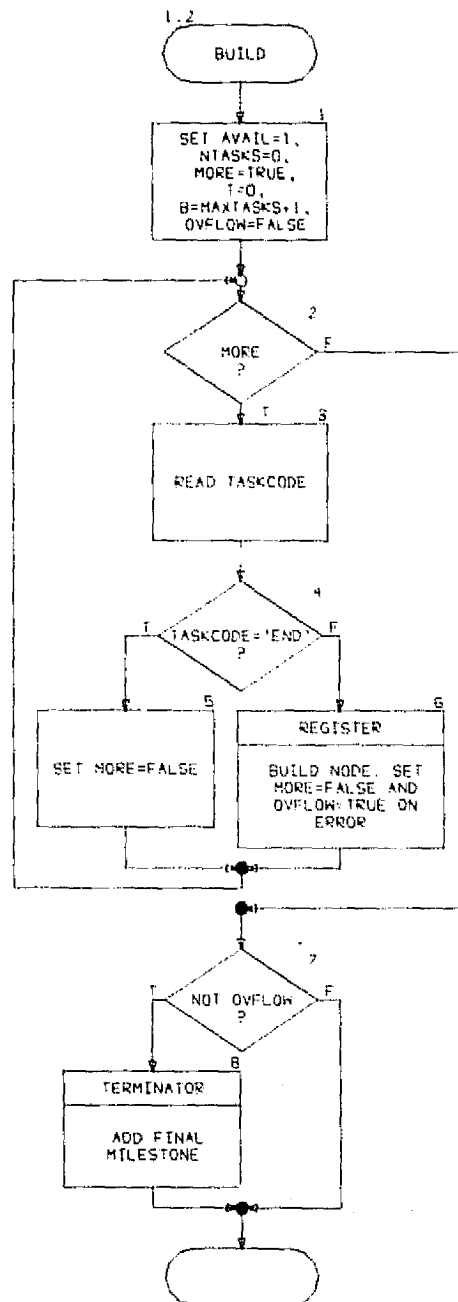
This procedure inputs the WBS information and builds the schedule network as described in Section 5.0.

On exit, the network has been formed. CODE, TITLE, DUR, COUNT, and TOP arrays contain the appropriate values, and SUC and NEXT arrays register successors, so as to simulate list format. NTASKS contains the number of tasks identified in the WBS.

- .1 AVAIL points to the next available SUC:NEXT pair to record successors. NTASKS is the Number of TASKS input. MORE is a boolean to control looping in steps 2 through 6, below. T is the integer index of the TSORT list (empty). B locates the Bottom of the unsorted list in TSORT (also now empty). OVERFLOW indicates that the network is not full.
- .2 MORE remains true while there are tasks to be input.
- .3 TASKCODE is the first field on the input line.
- .4 An "END" signals there are no more tasks,
- .5 thus, terminate the iteration when "END" appears.
- .6 As long as there are tasks, however, put these into the network. If there is no room in the node (i.e., task) arrays, signal "TASK OVERFLOW" error. If this task causes overflow of the successor linkage arrays, signal "LINKAGE OVERFLOW" error.
- .7 If there was no overflow,
- .8 add a termination milestone to the schedule network, if needed, linked as the successor to all tasks not having an explicit successor identified.

Example L-1 433

1.2
BUILD
4 APR 77



D:	<i>leg</i>	1/13/78
C:	<i>leg</i>	1/18/78
A:	<i>leg</i>	1/19/78

11 JAN 78

Chart Number	1.2.6
Module Name	REGISTER
Date	4/11/77

5.(1.2.6) REGISTER tasks into network

On entry, TASKCODE (string) contains the first field of the input record corresponding to a task. TASK (integer) and MOREPRED (boolean) have been declared, but do not contain pertinent information. The schedule network is either empty or partially completed. OVFLOW is global and false. The input medium is positioned just past the TASKCODE field of the current record.

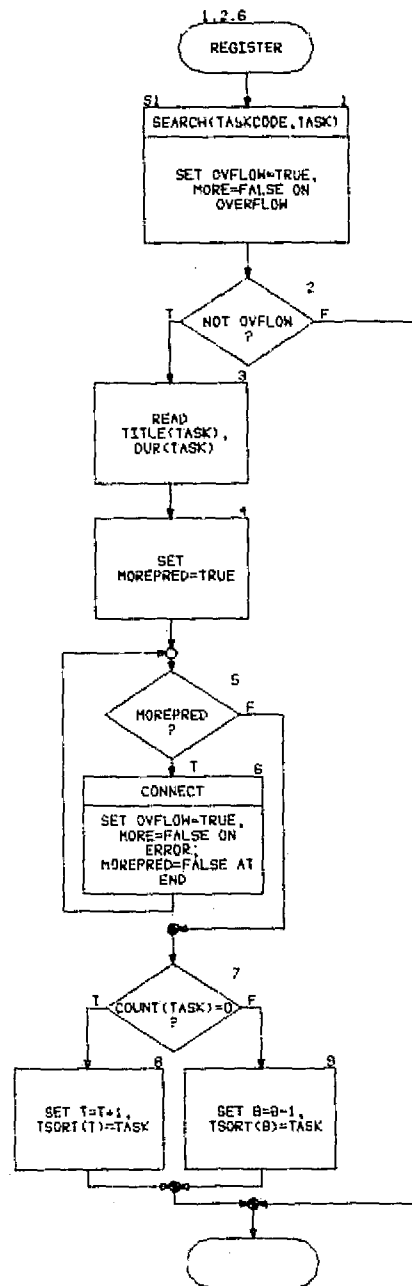
This procedure enters the task described on the input current record into the schedule network, if possible.

On exit, the task has been recorded into the network. If any tasks have been input with no predecessors named, these tasks are entered into the TSORT list (the first is presumed to be the project start milestone). OVFLOW and MORE will be toggled on error, and "TASK OVERFLOW" printed. T is the maximum index of tasks put into the TSORT list that are already in sort (no predecessors). The remainder of task indexes is inserted into TSORT from B to MAXTASKS, to aid in later scanning for the terminator.

- .1 SEARCH the network for the input TASKCODE identifier. If found, return its index in TASK; if not found, insert it into CODE at a blank location, and return the index in TASK. Set NTASKS to record the current Number of registered TASKs. Set OVFLOW true, MORE false on overflow, and print the error message "TASK OVERFLOW".
- .2 When there is room for the task,
- .3 input the title and duration, and
- .4 prepare to read list of predecessors by setting the MOREPRED flag true.
- .5 Then while there are predecessors on the record,
- .6 CONNECT these into the network by attaching a SUCCESSOR identifying this TASK to each named predecessor. If the SUCCESSOR array is depleted, set OVFLOW true and print "LINKAGE OVERFLOW". Set MORE and MOREPRED false to terminate processing. Increment the COUNT for this TASK for each predecessor.
- .7 However, if no predecessor was indicated, then
- .8 enter this task into bottom of the TSORT list, as these tasks are already sorted.
- .9 The other tasks are inserted at the top of the TSORT list to keep track of them for TERMINATOR (module 1.2.8).

Example L-1 435

1.2.6
REGISTER
11 APR 77



D:	1/13/78
C:	1/18/78
A:	4/9/78

41 JAN 78

Chart Number	1.2.6.6
Module Name	CONNECT
Date	4/11/77

5.(1.2.6.6) CONNECT task into schedule network

On entry, the TASK has been located, and its title, duration, and code have been entered into their corresponding array elements; remaining fields in the input record are predecessor task codes. MOREPRED is true, expecting such predecessors.

This procedure extracts these task codes, looks them up (inserting them into the network if not previously there), and causes a linkage between predecessor tasks and the current task by SUC: NEXT pairs. Error messages appear and OVFLOW and MORE are toggled on overflow.

On exit, all named predecessors will have been linked to the current TASK, and the number of predecessors will appear in the COUNT for that TASK. MOREPRED will be false when the list of predecessors has been exhausted.

- .1 Try to read a predecessor task code into the PRED string; however, if no predecessor is there, set MOREPRED false to terminate.
- .2 If there is a predecessor code named,
- .3 look up its index PTASK (responding to overflow if necessary).
- .4 If there was no overflow,
- .5 and if there is still room in SUC: NEXT arrays,
- .6 then link the current task as a successor to the predecessor task.
- .7 If there is no room in SUC: NEXT, emit the error message, and
- .8 set flags necessary to terminate operations.
- .9 If there is a task overflow, its message will have been printed by SEARCH. Hence, merely reset MOREPRED to terminate operations.

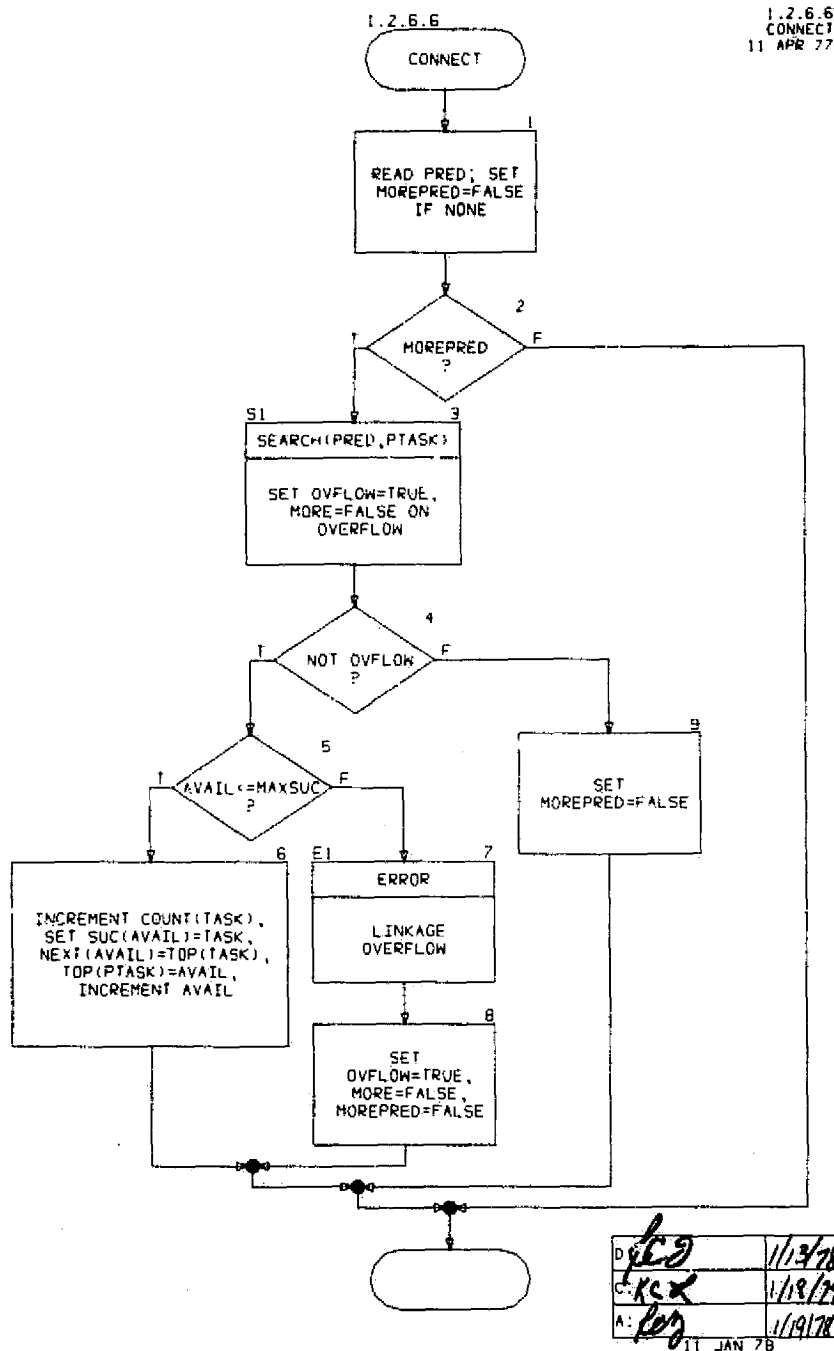


Chart Number	1.2.8
Module Name	TERMINATOR
Date	4/11/77

5.(1.2.8) Attach project TERMINATOR to network

On entry, the schedule network has been built, except that no project termination may have been provided. TSORT from 1 to T contains sorted task indices; from B to MAXTASKS, TSORT has recorded the other tasks. AVAIL points to the next available SUC: NEXT pair.

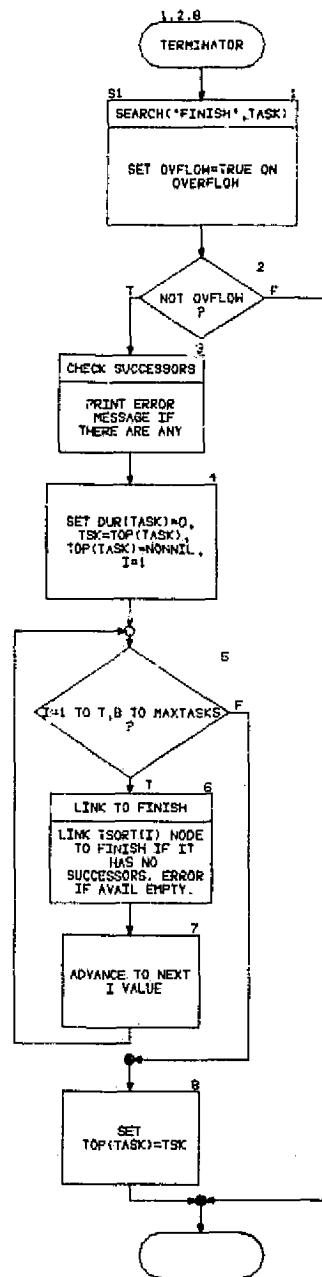
This procedure creates SUC: NEXT pairs and attaches them to every task not having any identified successors. If this causes an overflow, the appropriate error message appears. By sequencing through tasks as listed in TSORT, this algorithm is independent of the assignment of indices to tasks by SEARCH/S1.

On exit, the schedule network is complete, unless OVFLOW has been set true. The only task with no successor is the added termination node.

- .1 Find the "FINISH" task as TASK, or enter it into the network (responding to overflow if necessary), returning TASK. Set OVFLOW true and print "TASK OVERFLOW" if unsuccessful.
- .2 If there is no overflow (OVFLOW false),
- .3 assure that this TASK has no successors named by user. Inform user of erroneous usage, if detected, by printing the successor task code list.
- .4 Make a milestone of TASK. The TITLE is not altered (null if no title supplied). Save the finish milestone successor list TOP pointer (it should be nil), and change it to a non-nil value to prevent a SUCCESSOR linkage from being attached in step 6, below. (TOP(TASK) will be restored later in step 8). Initialize I to range through all tasks.
- .5 Loop through all tasks: first, the ones already in sort, and then the remainder. These are listed in the TSORT array from 1 to T and B to MAXTASKS.
- .6 Examine each task; if its TOP is nil, it has no successors, so get a SUC: NEXT pair and link this task to the FINISH milestone. Set OVFLOW true if no such linkage can be made. If TOP is non-nil, take no action.
- .7 Setting I to the "next" value is incrementing by 1 except when I goes beyond T, whereupon the "next" value is B. See step 5.
- .8 Reset the TOP of the termination milestone to indicate it has its previous successors (it was changed in step 4). In the anomalous case where this milestone has successors, this procedure will have created a cycle of tasks, which will then be detected by DIAGNOSE/1.5.7. Restoring TOP is necessary for correct diagnostic operation.

Example L-1 439

1.2.8
TERMINATOR
11 APR 77



D: Ray	11/13/78
C: KCK	11/18/78
A: Key	11/19/78

11 JAN 78

Chart Number	1.2.8.3
Module Name	CHECK_SUCCESSORS
Date	4/11/77

5.(1.2.8.3) CHECK SUCCESSORS of finish milestone

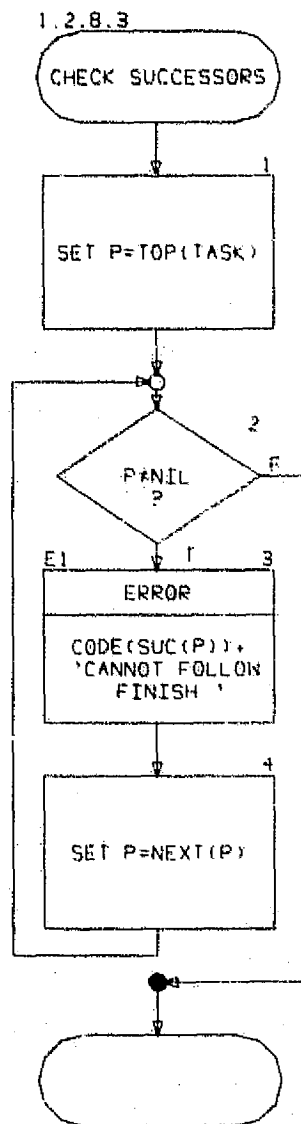
On entry, the FINISH task has been identified as TASK.

This procedure prints a list of task codes that listed the FINISH task code as a predecessor.

On exit, the data space is not altered.

- .1 The Pointer P, if non-nil, indicates that FINISH has appeared as a predecessor code of at least one task.
- .2 While there is a list of successors,
- .3 inform the user of the erroneous usage,
- .4 advance to the NEXT successor, and repeat.

Example L-1 441



1.2.8.3
CHECK SUCCESSORS
11 APR 77

D:	<i>leg</i>	1/18/77
C:	<i>KCR</i>	1/19/77
A:	<i>leg</i>	1/19/77

JAN 78

Chart Number	1.2.8.6
Module Name	LINK_TO_FINISH
Date	4/12/77

5.(1.2.8.6) LINK a task without successors TO the FINISH task

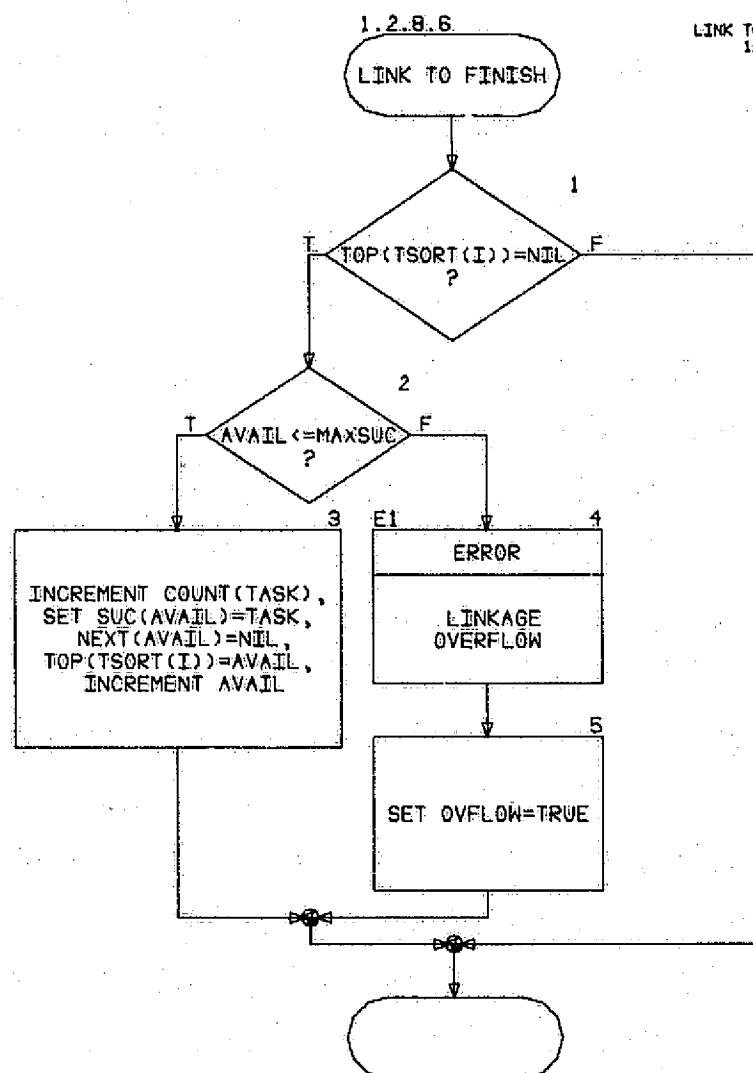
On entry, *I* is the index of the *TSORT* list of tasks where a task *TOP* is to be examined.

This procedure attaches a *SUC:NEXT* token to the *TOP* of the *TSORT(I)* task, if *TOP* is *nil* (or emits an error message and sets *OVFLOW* true if *AVAIL* has been exhausted), pointing to the *FINISH* task.

On exit, the *TSORT(I)* task is linked to the *FINISH* milestone if it had no successor, or else *OVFLOW* has been set true.

- .1 *TSORT(I)* is a task, so its *TOP*, if *nil*, indicates that the task had no stated successors.
- .2 Therefore, if there is a *SUC:NEXT* pair available,
- .3 attach the termination milestone as the successor to the *TSORT(I)* task.
- .4 But if there is no more room, output the error message, and
- .5 indicate that overflow has occurred.

Example L-1 443



D: <i>Ray</i>	1/13/78
C: <i>KCA</i>	1/19/78
A: <i>Ray</i>	1/19/78

11 JAN 78

Chart Number	1.5
Module Name	TOPOSORT
Date	4/12/77

5.(1.5) TOPOlogical SORTing of tasks

On entry, the network has been built. The TSORT list, which will contain the sorted tasks, has all tasks with no predecessors already in the list, in positions 1 through τ (the Tail of the list).

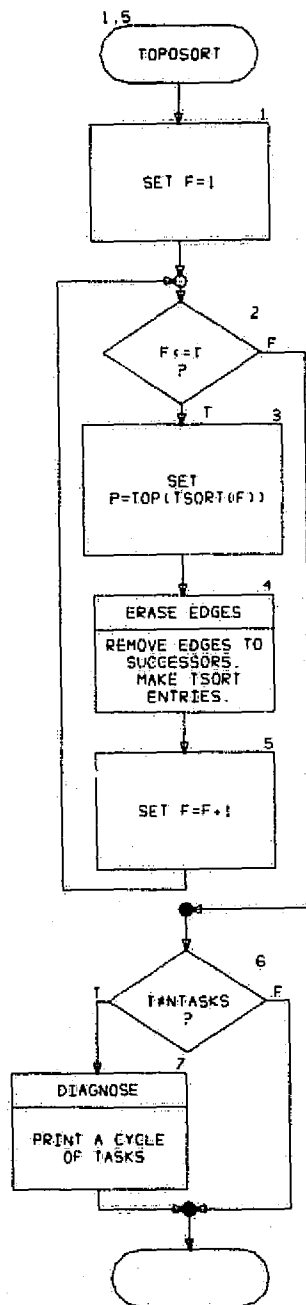
This procedure adds the remaining tasks to TSORT in topologically sorted order, unless the schedule network contains erroneous circular task requirements. In this latter case, the set of tasks not in the list and one schedule circular path are identified.

On exit, the COUNT array will have been destroyed. If an error was present in the schedule, the value of T will be less than NTASKS, which will cause SCHEDULER/1 to bypass the generation of the schedule.

- .1 F indexes the TSORT array from the Front
- .2 Iterate while there are tasks in the TSORT list which have not had their successor edges "removed."
- .3 P after this step Points to the SUC: NEXT list for a task which has no predecessors, or has had its predecessors "removed" in a previous iteration.
- .4 Reduce the COUNT field of all successor nodes of this Front task.
- .5 Having exhausted the successor list for a given task, advance the Front of the TSORT list to unqueue the next zero-predecessor task during the next iteration (step 2, above). Cycling does not occur when F goes beyond τ .
- .6 If the number of tasks deposited in TSORT is not equal to the number of tasks entered into the program, then
- .7 print a diagnostic message, "WBS HAS AT LEAST ONE CYCLE AMONG TASKS", followed by a list of task codes not in the TSORT list. Then determine and print one such cycle of tasks.

Example L-1 445

1.5
TOPOSORT
12 APR 77



D:	leg	1/12/78
E:	KCA	1/18/78
A:	leg	1/19/78

11 JAN 78

Chart Number	1.5.4
Module Name	ERASE_EDGES
Date	4/12/77

5.(1.5.4) ERASE EDGES to successor nodes

On entry, *P* has been set to the *TOP* field of the *TSORT(F)* task, *F* being the index into the *Front* of the *TSORT* list of node indices.

This procedure iterates through the *SUCCESSOR* list of the *TSORT(F)* task, reducing the *COUNT* field of each successor node.

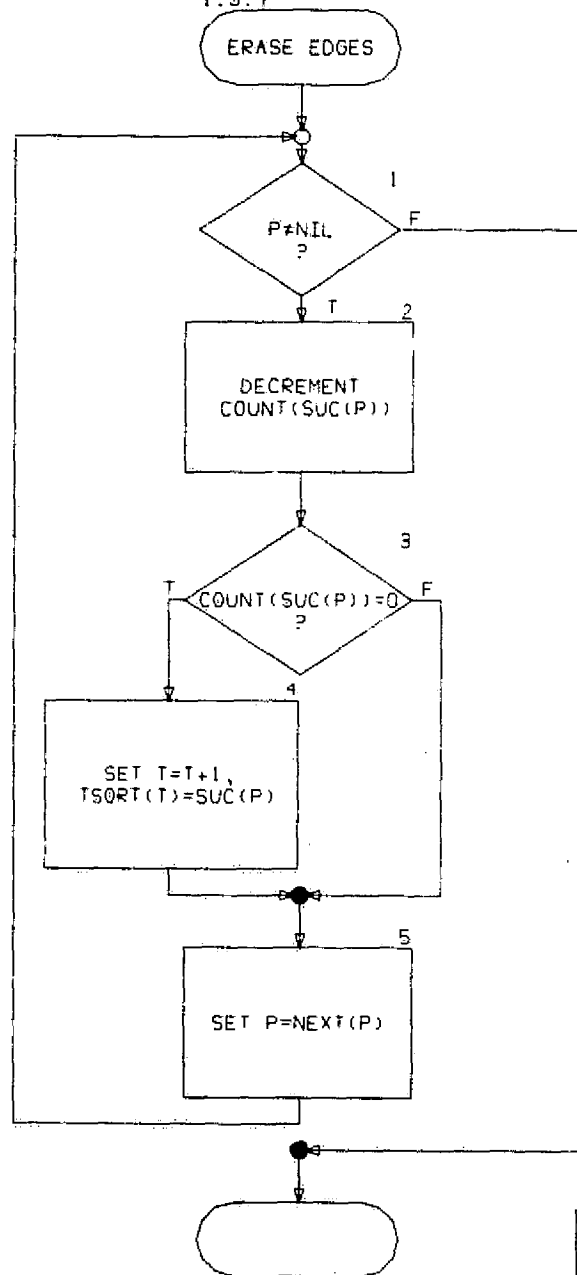
On exit, the *COUNT* fields of all successor nodes of the *F*-indexed *TSORT* will have been reduced by one.

- .1 Initiate an iteration for the current front task, and maintain this iteration while this task has successors (indicated by a valid *Pointer P*).
- .2 *SUC(P)* is a successor task; hence reducing the *COUNT* for this task constitutes "erasing" the edge, because
- .3 when all edges into a task have been erased, its *COUNT* will be zero, so
- .4 insert this task into the *TSORT* list, and adjust the *Tail T* accordingly.
- .5 Follow the successor list to the *NEXT* item, and repeat (as long as there are valid successors).

Example L-1 447

1.5.4

1.5.4
ERASE EDGES
12 APR 77



D:	Reg	1/13/78
C:	KCL	1/18/78
A:	Reg	1/19/78

11 JAN 78

Chart Number	1.5.7
Module Name	DIAGNOSE
Date	4/12/77

5.(1.5.7) DIAGNOSE the WBS network

The module described here is at present a STUB.

On entry, only $T < NTASKS$ have been entered into the TSORT list. Those tasks not listed in TSORT have non-zero COUNT entries.

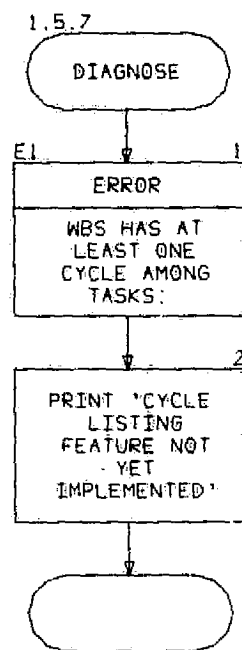
This procedure currently only emits a diagnostic message and notification that this feature is not yet implemented.

On exit, then, there is no change in the data space.

On completion, this module will print the task codes for all tasks having a non-zero COUNT. Not all of these are necessarily in a loop, but there is at least one loop among them. This module will then print "ONE SUCH LOOP IS", followed by the task codes of a circular path in the WBS, as determined by the algorithm on page 543 of Knuth (Reference 7.2.9).

Rationale: The features of this stub represent user enhancements in the form of exception handling. The mainline program features can be checked using a stub at this point.

Example L-1 449



1.5.7
DIAGNOSE
12 APR 77

D:	<i>Key</i>	11/2/78
E:	<i>KSA</i>	1/18/78
A:	<i>Key</i>	1/14/78

JAN 78

Chart Number	1.7
Module Name	DATES
Date	4/14/77

5.(1.7) Compute schedule DATES

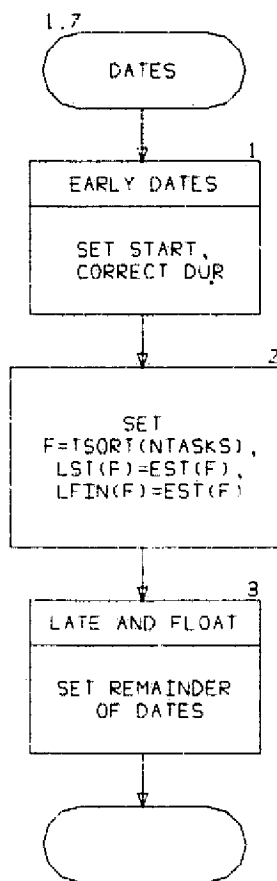
On entry, the schedule network has been built (COUNT is no longer used). The topological sort of tasks is in TSORT, with beginning task as task TSORT(1), and completion as TSORT(NTASKS). The starting task date (day of year) is contained in the DURATION field. All start, finish, and float times are zero.

This procedure scans TSORT in the forward direction to compute early start and finish times; then in reverse, to compute late start and finish times, plus the float.

On exit, the task attributes EST, EFIN, LST, LFIN, and FLOAT have been computed, and DUR of the starting milestone has been corrected (to zero). START contains the start milestone day of year. All other times are days past START.

- .1 Set START to the start milestone DURATION and zero this DUR. Then compute EST and EFIN for all tasks.
- .2 Now equate the Latest START and Latest FINish times to the Earliest START time (equal also to the earliest finish time) of the completion milestone. This is necessary for the reverse scan in the next step.
- .3 Finally, compute the LST, LFIN, and FLOAT times for all tasks.

Example L-1 451



1.7
DATES
14 APR 77

D:	Ken	1/13/78
C:	PCR	1/18/78
A:	Reg	1/19/78

11 JAN 78

Chart Number	1.7.1
Module Name	EARLY_DATES
Date	4/14/77

5.(1.7.1) Compute EARLY start and finish DATES

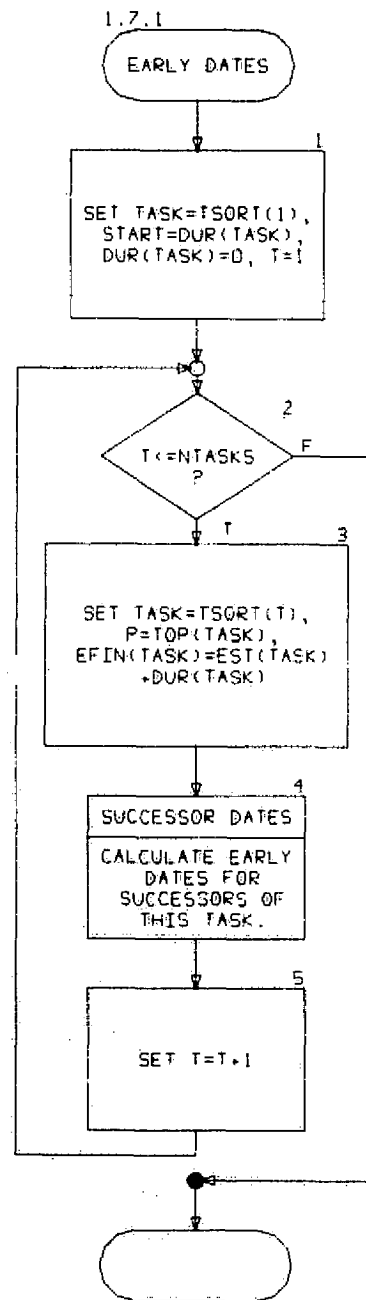
On entry, the state is identical with DATES/1.7, i.e., the network is built; TSORT contains the topologically sorted list, and all schedule dates are missing. The DUR attribute of the start milestone contains the starting date (day of year).

This procedure computes the early dates and corrects the DUR of the start milestone.

On exit, the DURATION of the start milestone is zero, and EST and EFIN for each task have been computed. START contains the start milestone day of year.

- .1 Transfer the start date to START and correct the DURATION. The procedure uses T to index through Tasks in the TSORT list in forward order. Begin the iteration at T=1,
- .2 and continue through T=NTASKS (the number of tasks).
- .3 For the current TASK, Point to the successor list, and fill in the Earliest FINish time. The Earliest STart time for this task will have already been computed, either by initialization (starting task) or by step 4, below.
- .4 For each of the successors of the current TASK, record its earliest start time as the early finish time of the current task when less than that previously recorded.
- .5 Then advance T to pick up the next task in the TSORT list, and repeat until all tasks have been processed.

Example L-1 453



1.7.1
EARLY DATES
14 APR 77

D:	Reg	1/13/78
C:	IC	1/18/78
A:	Reg	1/19/78

11 JAN 78

Chart Number	1.7.1.4
Module Name	SUCCESSOR_DATES
Date	4/15/77

5.(1.7.1.4) Record SUCCESSOR early start DATES for current task

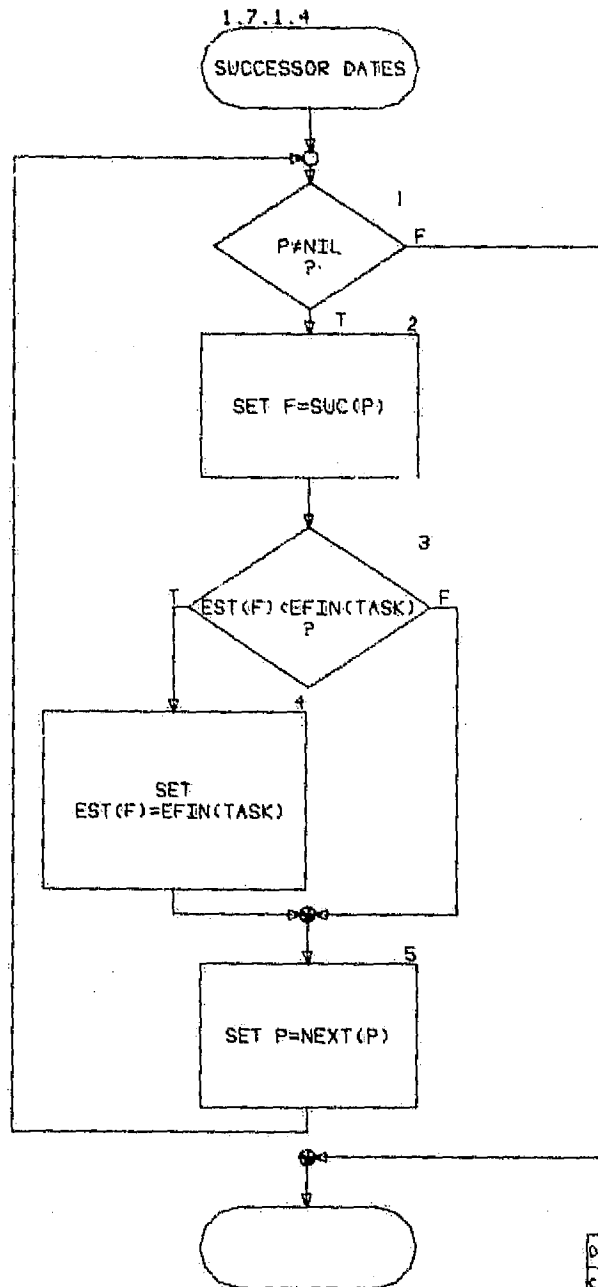
On entry, P points to the TOP of the current TASK, and the EFIN of this task has been computed.

This procedure sequences through the successor list (if any) of the current TASK, and enters EFIN as the earliest start (EST) of successor tasks whenever this EFIN is larger than the EST of that successor node.

On exit, all successors of the current TASK have been notified of the schedule constraints placed by that TASK.

- .1 Begin a loop to sequence through the successor list. During this iteration
- .2 let F be a successor of the current task.
- .3 The Earliest Start time of the successor must not precede the Earliest Finish of this TASK, so if such a requirement had previously been entered (it was initially zero),
- .4 update the successor Earliest Start appropriately.
- .5 Advance to the next successor in the list, if any, and repeat until the list of successors is exhausted.

Example L-1 455



1.7.1.4
SUCCESSOR DATES
15 APR 77

D:	Reg	1/13/78
C:	RS	1/19/78
A:	Reg	1/19/78

01 JAN 78

Chart Number	1.7.3
Module Name	LATE_AND_FLOAT
Date	4/14/77

5.(1.7.3) Compute LATE AND FLOAT dates

On entry, the early start and finish times for each task have been computed, and the starting milestone DURATION reset to zero. TSORT contains the topologically sorted list of NTASKS items.

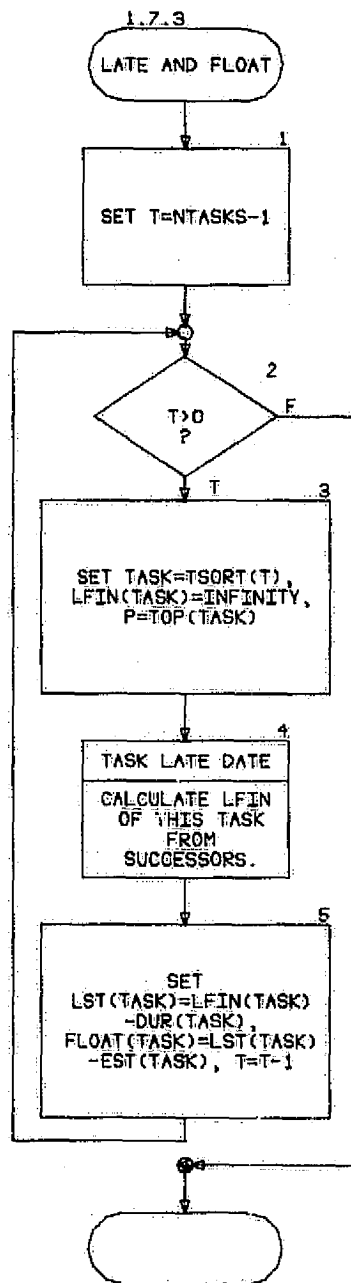
This procedure scans the TSORT list in reverse to compute late start/finish dates and the float time.

On exit, these quantities have been deposited in LST, LFIN, and FLOAT.

- .1 Set T to locate the next-to-last item in the TSORT list (whose successor list must be the project completion).
- .2 Initiate an iteration backward through the TSORT list (indexed by T), and continue until the list has been processed.
- .3 Pick up the current TASK, preset its Latest FINish to machine infinity, and point to the list of successors. Since LFIN for this TASK is to take the minimum value of the latest start time of its successors, the initial value of infinity is needed.
- .4 Iterate through the list of successors of this TASK (if any) and record the LFIN of this task as the LST of the successor task when this LST is smaller than the LFIN so far recorded.
- .5 After LFIN has been found, compute the Latest Start by subtracting the DURATION, and compute the float as shown. Finally, decrement T to pick up the next element in TSORT, and repeat until TSORT has been completely processed.

Example L-1 457

1.7.3
LATE AND FLOAT
14 APR 77



D:	10/13/78
E:	11/18/78
A:	11/18/78

11 JAN 78

Chart Number	1.7.3.4
Module Name	TASK_LATE_DATE
Date	4/15/77

5.(1.7.3.4) Record current TASK LATE starting DATE from successors

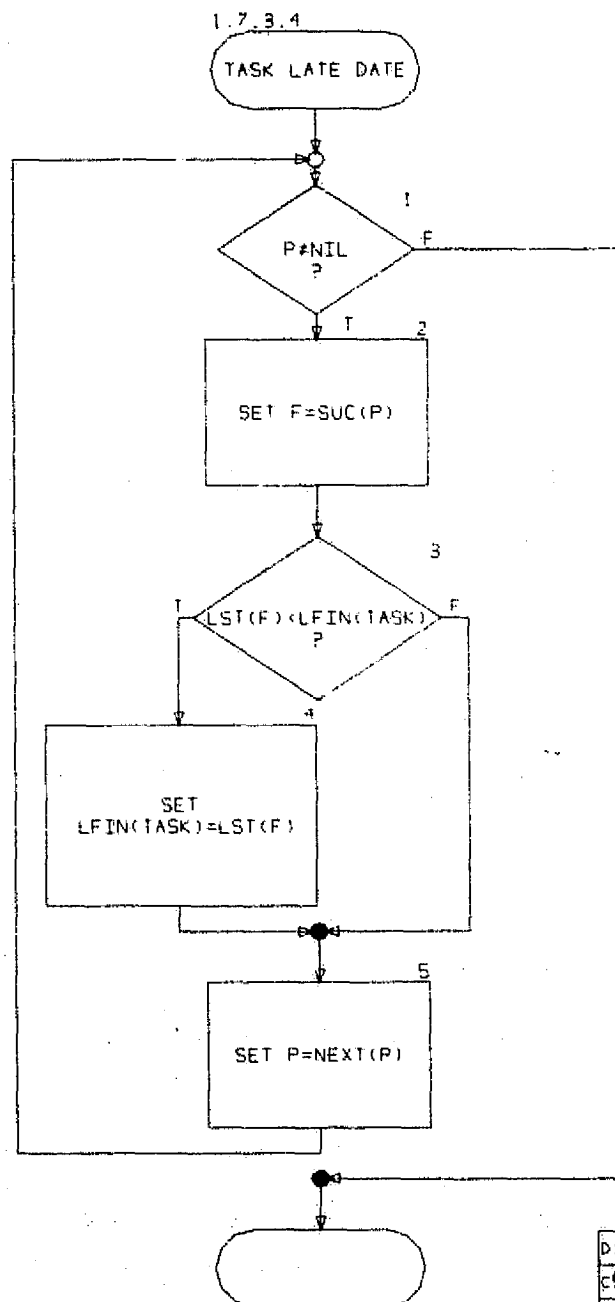
On entry, P points to the list of successor packets of the current TASK. LFIN of the current task has been initialized to infinity. The LST values of all successor nodes have already been computed due to the in-process reverse scan through the TSORT list.

This procedure sequences through the successor list (if any) of the current TASK, and sets the LFIN of this TASK to the LST of successor tasks whenever that LST is smaller than the current LFIN.

On exit, the LFIN for the current task has been established.

- .1 Prepare to iterate through the list of successors.
- .2 Pick up a successor task F.
- .3 The Latest FINish time for the current TASK must not exceed the Latest START of a successor; hence, if it does,
- .4 correct that situation as shown.
- .5 Then pick up the index of the next successor and repeat until the successor list is exhausted.

Example L-1 459



1.7.3.4
TASK LATE DATE
15 APR 77

D:	leg	1/12/78
C:	KCR	1/19/78
A:	leg	1/19/78

11 JAN 78

Chart Number	1.8
Module Name	DISPLAY
Date	4/14/77

5.(1.8) DISPLAY schedule report

On entry, TSORT contains a list of tasks in topologically sorted order, and the task early/late start/finish times have been computed. START contains the day of year of the start milestone. HEADER contains the top-of-page text to be used.

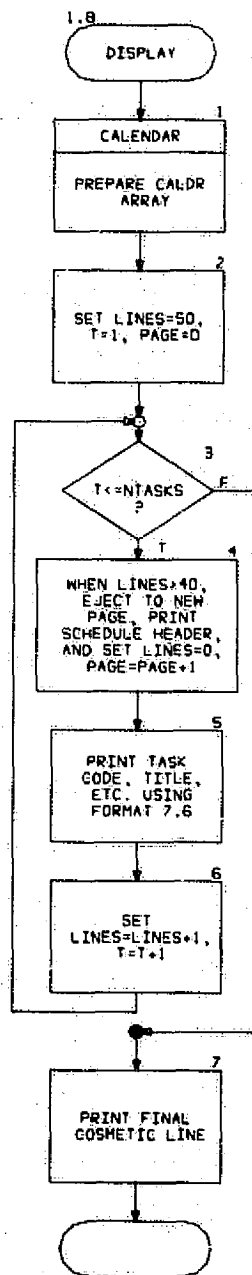
This procedure prints the schedule specified in format 7.5.

On exit, the entire data space is no longer required.

- .1 Read in the calendar data base so that the START date appears as CALDR(0) and CALDR(n) contains the date of the *n*th workday following START. If the project finish goes beyond the calendar or beyond MAXDATE elements, print an error message, but do not abort; in the former case, insert a "DAY *n*" designator into CALDR.
- .2 LINES will count the number of tasks output on a page, and T will index through the TSORT list. Preset these for later steps.
- .3 For each of the tasks 1, ..., NTASKS,
- .4 test whether the end-of-page is near. If so, advance to the top of the next page. If a top-of-form character is available, that can be used; if not, print line feeds to space correctly. Indicate that no tasks have yet been printed on this new page, advance the PAGE count, and print the header at the top of the new page, using the data prepared in HEADER.DATA/1.1.3.
- .5 Print the task CODE, TITLE, DURATION, EST, EFIN, LST, LFIN, and the corresponding dates via the CALDR array. Print the FLOAT time. If dates exceed the calendar, or if the project exceeds MAXDATE in duration, print "DAY *n*". Print "*" in column 1 if the float of this task is zero. See 7.5 for format details.
- .6 Then indicate that a line on the report has been used, advance T forward in the TSORT list, and repeat until the tasks have all been printed.
- .7 Print the final cosmetic lines across the report and terminate.

Note: Some implementations of this procedure may elect to sort the TSORT list on CODE, TITLE, EST, LST, EFIN, or LFIN as a step 0 prior to the procedure as given above. A suitable algorithm is given as Example 7.3.3.1 in Chapter 7 of Reference 7.2.10, but the definition of "elements out of order" must be reversed.

Example L-1 461



1.8
DISPLAY
14 APR 77

D: <i>kes</i>	11/3/78
C: <i>kes</i>	1/18/79
A: <i>kes</i>	1/19/79

11 JAN 78

Chart Number	1.8.1
Module Name	CALENDAR
Date	4/15/77

5.(1.8.1) Read in CALENDAR file

On entry, START contains the integer day-of-year on which the project starts. The calendar file contains a set of records beginning with the current year. The *n*th record corresponds to the *n*th day of the year, and contains two fields: The first is a boolean value (true if that day is a workday, false otherwise), and the second is a string denoting the date.

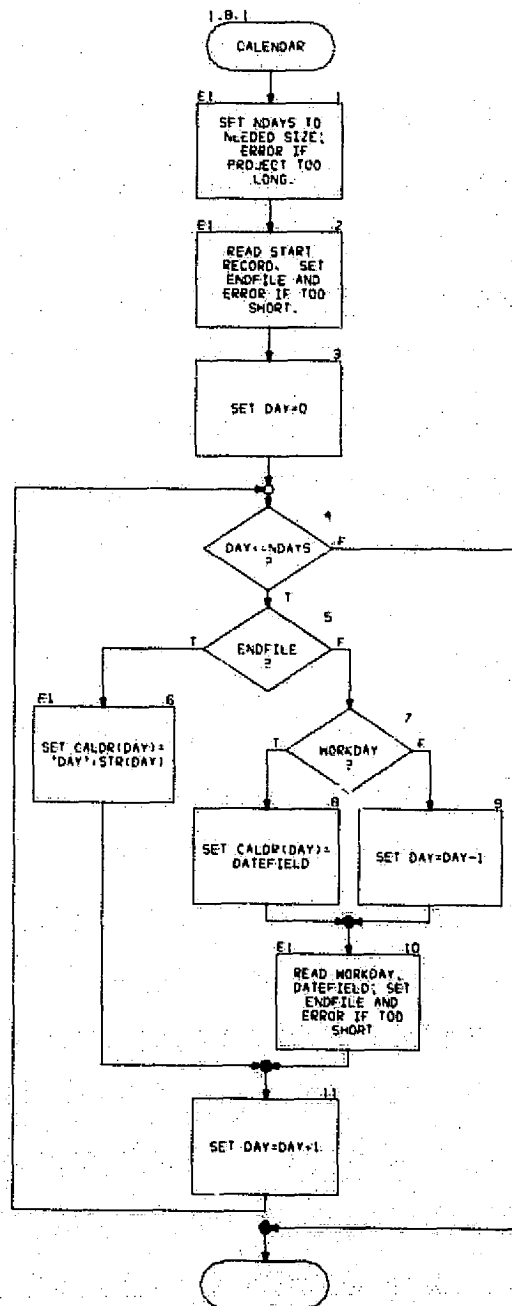
This procedure fills CALDR with LFIN(NTASKS) dates having a true workday flag. If the calendar file is too short, fill with "DAY *m*", where *m* is the project day, and print an error message. If the project goes beyond MAXDATE, also signal an error.

On exit, CALDR has been prepared, and, possibly, error messages have been emitted.

- .1 Set the integer variable NDAYS to the Number of project DAYS required, i.e., LFIN(NTASKS), or to MAXDATE, whichever is smaller. If LFIN(NTASKS) > MAXDATE, print the message "PROJECT TOO LONG".
- .2 Advance the calendar file and read the START-date record. Set the ENDFILE flag false if this can be done; otherwise set ENDFILE true and print the error message "CALENDAR TOO SHORT".
- .3 Set up to read the calendar file, starting at DAY zero,
- .4 and iterate through NDAYS.
- .5-.6 The project day number converted to a string becomes the CALDR entry after the endfile condition occurs.
- .7-.8 The DATEFIELD string read in from the calendar file becomes the CALDR entry if the day has a true WORKDAY flag.
- .9 If not a workday, decrement the day count to counterbalance the later incrementation in step 11, so as not to advance the project day count.
- .10 Read the next calendar file record; print the error message "CALENDAR TOO SHORT" and set ENDFILE true if reading hits an endfile condition on the attempted read.
- .11 Advance the day count, and iterate until CALDR contains the required number of days.

Example L-1 463

I.B.1
CALENDAR
15 APR 77



D: *Pen* 11/3/78
 C: *KC* 1/19/78
 A: *CS* 1/19/78
 11 JAN 78

Chart Number	E1
Module Name	ERROR
Date	4/18/77

5.(E1) ERROR handling routine

On entry, some error condition has been detected, and a message has been passed to this routine as the sole parameter. The parameter is herein treated as a string; however, the parameter may be coded as the integer index into an error message array, if all calls are properly annotated.

This routine prints "****ERROR****" followed by carriage return, line feed, and the parameter message.

On return, there has been no change in the program data space state.

No flowchart of this procedure is given.

Called from:

- 1.2.6.6.7
- 1.2.8.3.3
- 1.2.8.6.4
- 1.5.7.1
- 1.8.1.1 (implicit)
- 1.8.1.2 (implicit)
- 1.8.1.10 (implicit)
- S1.10

Chart Number	F1
Module Name	STR
Date	4/18/77

5.(F1) STRing function

This function converts its integer argument value to a string value. The returned string is the character representation of the integer, left and right justified (i.e., no spaces). No decimal or commas appear in the format. If the input value is positive, the output string is unsigned; if negative, the output is preceded by a minus sign.

No flowchart of this procedure is given.

Called from:

1.8.1.6

Chart Number	S1
Module Name	SEARCH
Date	4/18/77

5.(S1) SEARCH for task given task code

On entry, two arguments have appeared. The first is the task code TCODE, and the other is the corresponding task number TSK to be returned. The current number of tasks is NTASKS, and this number of task codes appears in the schedule network. The overflow flag OVFLOW is false; all data except the parameters are global.

This routine searches through the task codes already entered. If TCODE is found, the corresponding number is set into TSK. If not found, but room exists, then NTASKS is advanced one, and TCODE is made the task code of a new network node. If not found and there is no room, the OVFLOW flag is set true, MORE is set false, and the diagnostic message, "TASK OVERFLOW" is printed.

On return, either TSK has been set to the asked-for task number and OVFLOW is false, or else OVFLOW has been set true, in which case TSK is not meaningful. MORE will have been set false on overflow.

Stub rationale: Although slow and probably unsuitable for full operational use, the linear search/insert procedure is sufficient for checking the algorithms in the remainder of the program. On completion, this routine may well take the form of a hash table lookup of TCODE in the CODE array, as the program is not sensitive to the order in which tasks appear in the network.

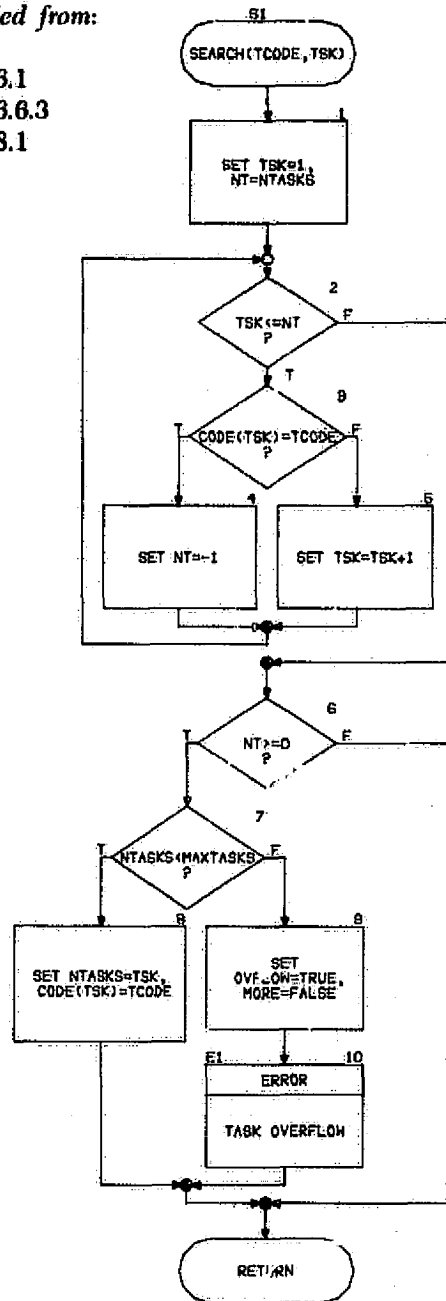
- .1 Initialize TSK and NT for searching the CODE array,
- .2 and search until the task is found or until all current codes have been examined.
- .3-.4 When the CODE at TSK agrees with that sought, stop the search.
- .5 Otherwise, increment the TSK index to continue searching.
- .6 NT will be zero or larger if no match was found.
- .7 Thus, if there is room to insert a new task code,
- .8 increment NTASKS (TSK is NTASKS + 1 by now), and enter TCODE into CODE at that point.
- .9 If there is no room for another task code, signal overflow and no more action required,
- .10 and print an error message.

Example L-1 467

Called from:

1.2.6.1
1.2.6.6.3
1.2.8.1

S1
SEARCH(TCODE,TSK)
18 APR 77



D: <i>Reg</i>	1/13/78
G: <i>K. J.</i>	1/18/78
A: <i>Reg</i>	1/19/78

11 JAN 78

5.2 Data Structure Definitions

Data structures defined in this specification are considered by this design to be globally accessible, so as to promote codability in almost any host language. Coding in languages that permit localized scoping and limited accessibility to data may arrange such declarations so as to contribute to more reliable programming practice.

Variables and constants used in this program are defined in the glossary (7.1). Amalgamation of arrays to form the schedule network is discussed in Section 5.0.

6. TEST AND VERIFICATION SPECIFICATIONS

6.1 Production Testing

During the coding activity that implements this program, input data shall be used to drive the program through every "flow line" of the procedures at least once. WBS networks shall be input to (singly) violate each of the network boundary conditions. Such data shall include:

- a. Empty WBS.
- b. WBS consisting only of "END" record.
- c. WBS not ending with "END" record.
- d. More than MAXTASKS tasks in the WBS.
- e. More than MAXSUC linkages (predecessors) in the WBS.
- f. Tasks with no predecessors.
- g. Tasks with no successors.
- h. FINISH task with successors.
- i. WBS has a circular set of tasks.
- j. WBS of moderate size (more than one output page) formatted correctly and within boundary constraints.

6.2 Acceptance Test Specifications

The program described herein shall be deemed suitable for operations, provided acceptance tests demonstrate:

- a. Proper response to the tests in Section 6.1.
- b. Error-free performance on several WBSs ranging in size over the acceptable limits.

- c. The detection of all of nine calibration errors of a random nature inserted into the program using the test data in (a) and (b), above, in a special test version for assessing test adequacy.

6.3 Quality Assurance Measures

Quality Assurance personnel shall audit the code resulting from this specification and attest that such code is a faithful translation of the documented procedures, that programming standards have been adhered to, and that all machine-dependent considerations conform within the limits specified herein. Such personnel shall then conduct or closely monitor the acceptance tests specified in Section 6.2, above.

7. APPENDICES

7.1 Glossary

This appendix contains the names of program modules, variables, constants, textual acronyms, and special terms used in this document.

AVAIL: AVAILABLE list pointer. A "pointer" into SUC:NEXT arrays. It contains the integer index of next available pair to be attached to the network. Value \leq MAXSUC.

B: Bottom of unsorted list. An integer index into the TSORT array. In REGISTER/1.2.6, the tasks from B to MAXTASKS are unsorted.

BUILD/1.2: Procedure name of the module that builds the schedule network.

CALDR: CALenDaR array. Array 0...MAXDATE of date, where date is either a string such as 15APR77 or integer such as 150477, which converts to a date in the form 15APR77 in 1.8.1.8.

CALENDAR/1.8.1: Procedure name of module that reads calendar file into CALDR array.

CHECK_SUCCESSORS/1.2.8.3: Procedure name of module that prints a list of task codes which have FINISH as a successor.

CODE: Task CODE array. The string array, ten characters by MAXTASKS, indexed by TASK, which contains task-code identifiers supplied to each task in WBS. Size \leq MAXTASKS.

CONNECT/1.2.6.6: Procedure name of the module that links predecessor tasks to the current task in the schedule network.

COUNT: The integer array indexed by TASK as part of the schedule network, which records the number of predecessors for the given task node. Size \leq MAXTASKS.

470 *Appendix L*

CPM: Critical Path Method. An algorithm for locating those tasks in a schedule whose slippage will cause the project termination date to slip also.

DATEFIELD: String variable to hold the calendar file work date read into the program.

DATES/1.7: Procedure name of module that computes late and early start and finish dates, and the float times for the schedule network.

DAY: An integer index denoting the project day beyond **START**. Day 0 is the project start.

DIAGNOSE/1.5.7: Procedure name of module to diagnose a schedule network having a loop.

DISPLAY/1.8: Procedure name of module that prints the schedule.

DUR: *DUR*ation. The integer array, indexed by **TASK**, which contains the durations of the referenced tasks in the schedule network, in days. Size \leq **MAXTASKS**.

EARLY_DATES/1.7.1: Procedure name of module that computes early start and finish dates for the schedule network.

EFIN: Early *FIN*ish time. The integer array, indexed by **TASK** as part of the schedule network, which contains the earliest finish times of tasks, reckoned in days from start. Size \leq **MAXTASKS**.

ENDFILE: Boolean flag set true when an endfile is found while attempting to read calendar file record.

ERASE_EDGES/1.5.4: Procedure name of module that "erases" edges to successor nodes by reducing the **COUNT** field of those nodes.

ERROR/EI: Subroutine that prints the string parameter passed to it as an error message.

EST: Early *ST*art time. The integer array, indexed by **TASK** as part of the schedule network, which records the earliest start time of tasks, reckoned in days from start. Size \leq **MAXTASKS**.

F: An integer index that locates the *Front* of the **TSORT** queue in **TOPOSORT/1.5**. It is also used to process **DATES/1.7**.

Float: The difference between latest and earliest start of a task in a schedule network.

FLOAT: An integer array indexed by **TASK** as part of the schedule network, to record float times, in days. Size \leq **MAXTASKS**.

HEADER: String information necessary for schedule header. See Section 7.5 for typical format of output; however, **HEADER** is not specified in detail in this document.

INITIALIZE/1.1: The procedure name of the module that declares all data structures and constants.

LATE_AND_FLOAT/1.7.3: Procedure name of module that computes late start and finish dates, and the float time for a schedule network.

LFIN: Latest *FIN*ish time. An integer array, indexed by *TASK* as part of schedule network, which records latest finish times of tasks, reckoned in days past start. Size \leq *MAXTASKS*.

LINES: An integer variable used to count the number of *LINES* put out on the current page.

LINK_TO_FINISH/1.2.8.6: The procedure name of a module that links a given node to the *FINISH* node if it has no successors.

LST: Latest *ST*art time. An integer array, indexed by *TASK*, as part of the schedule network, which records the latest start times of tasks, reckoned in days from start. Size \leq *MAXTASKS*.

MAXDATE: An integer program parameter. The maximum length of time that can be scheduled in a project. A compile-time constant.

MAXSUC: *MAX*imum number of *SUC:NEXT* pairs to be allowed in network. A compile-time integer constant.

MAXTASKS: *MAX*imum number of *TASKS*. A compile-time integer constant.

Milestone: An achievement signalled by the accomplishment of a task or set of tasks. In this program, a milestone is a zero-duration task linked to predecessor tasks in order to show achievement of all preceding tasks before advancing to a next set of tasks.

MORE: A boolean used for loop control. Set false by *SEARCH/S1* if task overflow occurs.

MOREPRED: A boolean loop flag, true while iterating upon predecessors of a node in *REGISTER/1.2.6*.

NDAYS: The integer Number of project *DAYS*, total duration or limited by *MAXDATE* in *CALENDAR/1.8.1*.

NEXT: A "pointer" array indexed by "pointers." An element contains the index of the next *SUC:NEXT* pair for successor nodes of the current task in the schedule network. Size \leq *MAXSUC*.

NIL: A compiler constant equal to zero, to represent the null pointer. The invalid integer index into *SUC:NEXT* arrays that indicates no further successors are present.

NONNIL: Any non-nil value of "pointer" into *SUC:NEXT* arrays. Used in *TERMINATOR/1.2.8*. A compile-time constant.

472 Appendix L

NT: An integer variable local to the stub SEARCH/S1, used to control the search. Initially set to the Number of Tasks, but reset to -1 if the sought task code is found.

NTASKS: An integer, the current Number of TASKS input into the schedule network. Value \leq MAXTASKS.

OVFLOW: A boolean variable, false unless task or successor linkage overflows occur.

P: Pointer. A "pointer" into SUC:NEXT arrays, used in various localized procedures.

PAGE: An integer value used by DISPLAY/1.8 to record the page number of the WBS report.

PERT: Program Evaluation and Review Technique. A method that aids in the planning, scheduling, monitoring, utilization, and reporting of project resources by means of task definition, work breakdown structures, resource constraints, and schedule networks.

Pointer: The index of SUC:NEXT array elements.

PRED: PREDeccessor task code. A string, which is a task code read in from a WBS record. Local to CONNECT/1.2.6.6.

PTASK: Predecessor TASK. An integer, the index of a predecessor task. Local to CONNECT/1.2.6.6.

REGISTER/1.2.6: Procedure name of the module that enters tasks into the schedule network.

SCHEDULER/1: The name of the main program.

SEARCH/S1: Subroutine to search for a given task code and return the task number; it inserts the task code, if not found, into the network, and sets OVFLOW as appropriate.

START: An integer variable that contains the project-start-milestone day-of-year read-in.

STR/F1: The STRing function, which converts an integer variable to a string variable. Used in CALENDAR/1.8.1.

SUC: SUCcessor. A "node" array indexed by "pointers" to denote edges in the schedule network. The "node" value is an index into task information arrays. Size \leq MAXSUC.

SUCCESSOR DATES/1.7.1.4: Procedure name of a module that calculates the EST restrictions placed on successors of the current TASK.

T: An integer index of the Tsort list. During sorting, the tasks in Tsort indexed 1, ..., T are in sort. Tasks beyond are probably not. $T \leq$ NTASKS.

Task: An item in a Work Breakdown Structure (WBS) characterized by work with definable inputs and outputs, predecessor and successor work constraints, probable or estimated duration, and assigned responsibilities.

TASK: An integer index into the task information arrays in the schedule network. $TASK \leq NTASKS$.

TASK_LATE_DATE/1.7.3.4: Procedure name of a module that calculates the LFIN of the current TASK from the LST of all its successors.

TASKCODE: A string, recording the task code input in BUILD/1.2. If "END" appears as a task code, then no more tasks will be input.

TCODE: A string parameter passed into the stub SEARCH/S1 having the value of the sought-for Task CODE.

TERMINATOR/1.2.8: Procedure name of the module that adds the "FINISH" milestone to the schedule network.

TITLE: A string array, 32 characters \times MAXTASKS, indexed by TASK, which contains the task title, as part of the schedule network. Size \leq MAXTASKS.

TOP: A "pointer" array indexed by TASK as part of the schedule network. TOP "points" to SUC:NEXT array elements to give successor information. Size \leq MAXTASKS.

TOPOSORT/1.5: Procedure name of module to sort the TSORT list topologically.

TSORT: Topological SORT list. An integer array indexed by τ , which contains the topologically sorted list. The first element is the task number of the start milestone. While being constructed, items up to τ are sorted. Items from B to MAXTASKS are tasks held for later query by TERMINATOR/1.2.8. Size \leq MAXTASKS.

TSK: An integer parameter passed from stub SEARCH/S1 having the value of the TaSK number corresponding to an input task code.

WBS: Work Breakdown Structure, or hierarchy of tasks refined into subtasks until subtasks are finite, manageable units.

WORKDAY: A boolean variable to hold the calendar file workday flag, true if the corresponding day is a workday, false otherwise.

7.2 References

- 1 Prather, R. E., *Discrete Mathematical Structures for Computer Science*, Houghton Mifflin Company, Boston, MA, pp. 220-278, 1976.

Mathematical background for schedule networks and the critical path method.

474 Appendix L

- .2 DoD and NASA Guide, *PERT/Cost Systems Design*, Office of the Secretary of Defense and the National Aeronautics and Space Administration, U.S. Government Printing Office, Washington, DC, June 1962.

A description of the PERT system application.

- .3 *Application Description Manual, 1130 Project Control System*, IBM Corp., White Plains, NY, 1968.

Introductory guide to the use of the IBM program, with some theory and examples.

- .4 Tausworthe, Robert C., *Standardized Development of Computer Software, Part II: Standards*, Chapter 16, this text.

Defines documentation levels.

- .5 *Ibid.*, Chapter 12.

Contains documentation standards and guidelines.

- .6 *Ibid.*, Chapter 13.

Contains coding standards and guidelines.

- .7 *Ibid.*, Chapter 14.

Contains testing standards and guidelines.

- .8 *Ibid.*, Chapter 15.

Contains QA standards and guidelines.

- .9 Knuth, D. E., *The Art of Computer Programming, Vol. I: Fundamental Algorithms*, Addison-Wesley Publishing Co., Reading, MA, pp. 262-265, 1969.

Gives the topological sorting algorithm used here.

- .10 Tausworthe, Robert C., *Standardized Development of Computer Software, Part I: Methods*, Prentice-Hall Publishing Company, Englewood Cliffs, NJ, 1977.

Gives the general methodology for top-down, modular, structured development of programs.

7.3 Potential Future Modifications

The extension of the program specified here into a full project planning, evaluation, and reporting tool is not covered by this specification. However, several candidate features are deemed necessary in such a tool. These are:

- a. Data-base operation, including update capability.
- b. Capability to assign resources (manpower, dollars) and to allocate facilities to individual tasks.
- c. Access to alternate calendars.
- d. Capability to enter required or actual dates for milestones.
- e. Options to print other types of reports, such as schedule (GANTT) chart, milestone data only, cumulative duration rate chart, resource expenditure profile, sorted schedule according to completion or start date, etc.

7.4 Error Messages and Diagnostics

This appendix contains a list of the error messages and conditions under which these are invoked, as specified by Sections 4 and 5 of this document. Other error messages may appear if host-dependent considerations dictate.

All error messages output by ERROR/E1 will appear preceded by

****ERROR****

Overflow errors terminate processing, whereupon the following message appears appended to the prefix above:

SCHEDULER TERMINATED

The detected errors are as follows:

CALENDAR TOO SHORT

Calendar file does not cover the number of days in the project.

task code CANNOT FOLLOW "FINISH"

The WBS has a FINISH task with a successor task identified by *task code*.

LINKAGE OVERFLOW

The schedule network has too many successor relationships (edges).

TASK OVERFLOW

The WBS has too many tasks (nodes).

476 *Appendix L*

WBS HAS AT LEAST ONE CYCLE AMONG TASKS

The schedule network contains a circular set of predecessor-successor relationships.

7.5 Output Report Format

The output report format is defined in Figure 7.5.1. The top line, giving the project name, is the HEADER record input by HEADER_DATA/1.1.3. The report date is assumed available via the operating system. The format displays the various field widths, and gives a typical example. Asterisks signal critical-path items.

END OF SPECIFICATION

WIDGET PROJECT
WORK BREAKDOWN STRUCTURE

DATE: 12 MAR 77

CODE	TITLE	DURATION	START		FINISH		FLOAT	
			EARLY	LATE	EARLY	LATE	TIME	DAYS
			DAY	DATE	DAY	DATE	DAY	DATE
*1	START	0	0	11APR77	0	11APR77	0	11APR77
4	ORDER PACKAGING MATERIALS	10	0	11APR77	35	31MAY77	10	25APR77
3	FABRICATE USING PARTS ON HAND	20	0	11APR77	10	25APR77	20	9MAY77
*2	PROCURE NEEDED PARTS	30	0	11APR77	0	11APR77	30	23MAY77
*5	FABRICATE USING PROCURED PARTS	15	30	23MAY77	30	23MAY77	45	14JUN77
*6	INTEGRATE AND TEST ASSEMBLY	20	45	14JUN77	45	14JUN77	65	14JUL77
*7	PRODUCT COMPLETION	0	65	14JUL77	65	14JUL77	65	14JUL77
*8	PACKAGE AND DELIVER ASSEMBLY	10	65	14JUL77	65	14JUL77	75	28JUL77
*11	TRAIN PERSONNEL	15	75	28JUL77	75	28JUL77	90	18AUG77
10	ASSEMBLE TEST GEAR	5	75	28JUL77	80	4AUG77	80	4AUG77
9	INSTALL AT CUSTOMER FACILITY	5	75	28JUL77	80	4AUG77	85	11AUG77
12	PERFORM READINESS TESTS	5	80	4AUG77	85	11AUG77	90	18AUG77
*13	PERFORM ACCEPTANCE TESTS	5	90	18AUG77	90	18AUG77	95	25AUG77
*14	FINISH	0	95	25AUG77	95	25AUG77	95	25AUG77
*FINISH		0	95	25AUG77	95	25AUG77	95	25AUG77

* CRITICAL PATH ITEM

Figure 7.5.1. Output format of the SCHEDULER program illustrated by a hypothetical set of tasks representing a "Widget Project"

Example L-1 477

EXAMPLE L-2

DECISION TABLE CONVERSION USING THE POLLACK PROCEDURE

1. INTRODUCTION

1.1 Purpose

This program converts a limited entry decision table (LEDT) input by a user to a CRISP program using the Pollack "optimization" procedure, based on a least-decision-time cost criterion.

1.2 Scope

This specification covers only machine-independent considerations relative to implementation. Standards and conventions, system environment and interfaces, test and verification details, and other lower-level considerations are not covered.

1.3 Applicable Document

Robert C. Tausworthe, Standardized Development of Computer Software, Part I, Standards, Chapter 8, Section 8.3.3 through 8.3.5, pages 258-271, Prentice-Hall, Englewood Cliffs, N.J., 1977 (Reference 5.1).

1.4 General Description

The application of LEDT's to generation of flowcharts, and therefore CRISP programs, is covered by the applicable document named above. The Pollack procedure herein described is a recursive algorithm for processing an LEDT composed of condition stubs and entries, action stubs and entries, costs associated with each condition, and probabilities associated with each rule. No "don't-care" entries are permitted in the LEDT input. The basic procedure, due to Pollack (Reference 5.2), converts an unambiguous LEDT into a flowchart (or CRISP program) as follows:

- a. Select one row of the original LEDT by a suitable criterion C (to be discussed later). The condition in that row becomes the first comparison of the flowchart.

- b. Decompose the table into two subtables having one less row - either subtable may perhaps only contain one action - and associate each subtable with a branch of the flowchart decision. That is, one subtable consists of all the remaining conditions and the set of rules for which the condition selected in (a) above is true; the other is similar, except that the condition is false.
- c. If a subtable has more than one action, select one of its rows by criterion C and attach the condition for that row to the proper branch of the previously selected condition producing that subtable.
- d. Continue (b) and (c) above on each subtable until each rule of the original LEDT is represented in a branch of a condition (or until a subtable indicates that the original table contained redundant or contradictory rules).

The criterion C, above, can, among other things, check for redundancy or contradiction among rules. If, at any stage, two rule columns exist without containing at least one Y-N pair in some row, redundancy or contradiction exists. If the actions for both rules are the same, the rules are redundant and one can be eliminated; if the actions are different, the rules are contradictory, and the table is in error.

The criterion C applied in this program (denoted as criterion C1 in the document referenced in Section 1.3 above) computes, for each condition to be tested in a given (sub)table, the decision-time-cost for that decision multiplied by the sum of probabilities of all rules r_1 and r_2 in the (sub)table such that $r_1 < r_2$, whose action entries are the same, and whose condition entries permit a "don't care" condition to replace a Y-N pair. This value for a given condition is known to be the expected extra time contributed to the execution of a program if that condition becomes the one selected in step (a) above. The criterion C used here chooses the condition so as to minimize this expected extra time.

The reader is referred to the cited applicable document for further information regarding the general method.

2. THE METHOD

The condition stub in an LEDT may be eliminated from processing if each of the condition entry Y's and N's can be tagged with the number of the condition to which it applies. Thus, a Y2 in the table would make it clear that a Y response to condition 2 applied at that point. The program described in Section 5 uses +i and -i to record Y and N answers to condition i, respectively. Such tagging is necessary in order to

interchange condition rows and rearrange rule columns in subtables of the LEDT.

Starting with the complete table, let the best condition be found according to the criterion C above. Suppose this were condition 3. Then interchanging condition rows 3 and 1 brings the best condition to the top of the table. Next, by exchanging columns of the LEDT appropriately, all columns with Y responses to the top condition can be made to appear to the left of all columns with N in the top row. The subtable with a Y answer to the top condition extends from column 1 on the left to the middle of the table on the right, and from condition row 2 on down. The same procedure may now be applied to each of the two subtables just formed.

In general, then, the method is the recursive application, starting at a TOP=1, LEFT=1, and RIGHT=2**n, of the following steps: (1) for each subtable, determine the best condition row between TOP and n, based on the rules from LEFT to RIGHT, (2) exchange this condition row with the TOP row between LEFT and RIGHT limits, (3) permute columns to arrange all Y answers on the TOP row to be left of all N answers, (4) determine the MID column as the first N response in the TOP row, and, finally, (5) initiate the optimization of two subtables with a new TOP of TOP+1, and boundaries (LEFT, MID-1) and (MID, RIGHT), respectively. Note that permuting columns of a subtable never changes the condition entries above TOP.

Each time a best condition is found, it is printed in an IF statement; the value of TOP indicates the level of nesting, and, thus, the indentation to be used. The left subtable corresponds to the THEN clause, and the right subtable to the ELSE clause. Hence, upon returning from processing the left table, an (ELSE) is printed. After processing the right subtable, an ENDIF completes the processing of that subtable.

Whenever a subtable possesses rules that all invoke the same action, then no condition needs to be tested; hence, a text which indicates the action is printed at that point, and no further processing of the subtable takes place.

3. FUNCTIONAL SPECIFICATIONS

This program as specified here accepts an LEDT from an unspecified medium in an unspecified format; however, the information input shall be equivalent to that shown in Figure 8-10 of the cited applicable document. These entries include the following:

- a. Number of conditions: an integer, n in Section 2, above.
- b. Condition text: strings which state the conditions appearing in the condition stub, one string per condition. These will appear in IF statements.
- c. Action text: strings which identify action sets invoked by rules. These will appear in THEN and ELSE clauses.
- d. An action assignment list, which matches indices of action texts to their proper rules, listed in a standard order.
- e. A probability list, which matches execution frequencies to their proper rules, listed in the same standard order of rules.
- f. Condition test times, which state the times required to test each of the given conditions.

Processing this input as described in the cited document produces an indented CRISP listing of the program consisting of nested IFTHENELSE structures. The condition text strings are placed in IF statements. When a rule is finally resolved by nested condition outcomes, the clause corresponding to that rule is the action text string given on input for that rule.

The following example serves to illustrate the application of this program and specify the output format. The example shown in Figure 8-10 of the cited document has

- a. The number of conditions (integer): 3
- b. Condition text (strings): C1, C2, C3
- c. Action text (strings): A1, A2, A3
- d. Assignment list (integers): 1, 1, 3, 2, 1, 2, 2
- e. Probability list (reals): .1, .15, .15, .2, .25, .05,
.05, .05
- f. Condition test times (reals): 50., 68., 25.

In (c) and (d) note that the action assignments and probabilities are not given in the same order of rules as shown in the referenced figure. So as to avoid having to input individual Y-N entries into the condition stub, the program assumes a standard condition stub, ranging from all-Y as rule 1, to all-N as rule 2**c, progressing in binary radix form with the bottom condition toggling every rule, etc.

The output for this example, corresponding to Figure 8-11 in the reference, is

```

IF (C2)
:  IF (C3)
:    :  A1
:    :->(ELSE)
:    :  IF (C1)
:    :    :  A1
:    :    :->(ELSE)
:    :    :  A2
:    :    :..ENDIF
:    :..ENDIF
:->(ELSE)
:  IF (C3)
:    :  IF (C1)
:    :    :  A3
:    :    :->(ELSE)
:    :    :  A2
:    :    :..ENDIF
:    :->(ELSE)
:    :  A2
:    :..ENDIF
:..ENDIF

```

4. PROGRAMMING SPECIFICATIONS

4.0 Overview

The major data structures required by the program are those required to form the elements of the decision table. In the specifications below, these table structures will be considered as global to all modules within the program, but only for convenience in describing the algorithms. Data declarations are not covered by the program descriptions, but are, of course, necessary once the programming language is chosen.

Subroutine and function argument parameters are assumed here to be local to a subroutine and available as global within that subroutine (i.e., available to subprograms). Thus, a parameter, say TOP, passed as an argument to the POLLACK subroutine at the main program level, is not the same location as the argument parameter, TOP, local to that subroutine and accessed by subprograms PRINT_ELSE and PRINT_ENDIF. Local variables are identified in the narrative discussions of each program module.

Input of the LEDT is handled by a subprogram ENTER_LED, which accepts information, as described in Section 3, above, from an arbitrary medium and formats it for the recursive subroutine POLLACK, which optimizes the table and prints the cosmetized translation into CRISP.

C-6

The global data structures are

N Integer number of conditions in the LEDT.

CONDITION String array holding condition stub textual descriptions, in order 1,...,N.

PROB Real array holding rule probabilities, for rules in standard order 1,...,2**N.

A Integer number of distinct action combinations.

ACTION String array holding textual descriptions of the action combinations which apply to the various rules, 1,...,A.

RULE Integer array holding indices of the action strings which apply to each of the various rules, r=1,...,2**N, in standard order.

LEDT Integer array, $N \times 2^{**}N$ holding the Y-N (or T-F) condition entries. The i, j th element is ic , where $+c$ corresponds to a true outcome of condition c , and $-c$ corresponds to the false outcome. Initially, $LEDT(i, j) = \pm i$; i.e., the i th row records the Y-N entries for the i th condition, for each rule $j = 1, \dots, 2^{**}N$. Since subtables are permuted by the program, the condition correspondences are traceable by the entry values.

TIMECOST Real array of times required for making decisions, for each condition 1,...,N.

Subtables of LEDT required by the FOLLACK subroutine are defined by local parameters TOP, LEFT, and RIGHT.

Chart Number 1
Module Name LEDT CONVERSION
Date 5/27/77

4. (1) Main Program, LEDT CONVERSION

Program: LEDT CONVERSION <*27 May 77*> MOD# 1
.1 do ENTER LEDT <*also set N to number of conditions*>
.2/S1 call POLLACK(1, 1, 2**N) <*optimize LEDT from
 <*TOP=1 to bottom, LEFT=1 to RIGHT=2**N*>
 endprogram

On entry, a file of data exists containing the LEDT data.

This program reads the file into a structure modelling the LEDT, optimizes it using the Pollack procedure based on minimized execution time, and prints a CRISP listing of the LEDT translation.

- .1 Initialize the data structures LEDT, RULE, ACTION, A, PROB, CONDITION, and N with data input as directed by the user.
- .2 Optimize the entire table, bounded at the top by condition row 1, on the left by rule column 1, and on the right by rule column 2**N. Print a CRISP cosmetized listing of the translation. If there is an error in the input which results in an ambiguous subtable, print an error message, "table entered in error".

Module Number 1.1
Module Name ENTER LEDT
Date 6/15/77

4. (1.1) Table entry subprogram, ENTER LEDT

On entry, the program is uninitialized.

This procedure declares and fills the global data structures A, ACTION, N, CONDITION, RULE, and PROB with data input from a user-specified source. The LEDT array is sized using N and then filled with the standard initialization described in Sections 3 and 4.0.

On exit, all of the above data structures are initialized as required.

The procedure for this module is not covered by this specification. It may conceivably accommodate a user at an interactive terminal, input from a file or deck of cards in batch mode, or a combination of the two. For this reason, such procedure is not defined here. However, in any case, having executed, this module performs the indicated initializations so that the remainder of the program is insensitive to the input mode.

486 Appendix L

Module Number S1
Module Name POLLACK
Date 5/27/77

4. (S1) Table processing subroutine, POLLACK

```
Subroutine: POLLACK(TOP, LEFT, RIGHT) <*27 May 77*>    MOD# S1
.1      if (LEFT = RIGHT)
.2/S2   :   call PRINT_ACTION(LEFT, TOP)
.3      :->(TOP <= N AND LEFT < RIGHT)
.4/S1   :   call CHECK_RULES(LEFT, RIGHT, SAME)
.5      :   if (SAME)
.6/S2   :       call PRINT_ACTION(LEFT, TOP)
.7      :       :->(else)
.7/F1   :       let B=BEST(TOP, LEFT, RIGHT)
.8/S4   :       call EXCHANGE_CONDITIONS(TOP, B, LEFT, RIGHT)
.9/S5   :       call PARTITION_RULES(TOP, LEFT, RIGHT, MID)
.10/S6  :       call PRINT_CONDITION(TOP, LEFT)
.11/S1  :       call POLLACK(TOP+1, LEFT, MID-1)
.12     :       do PRINT_ELSE
.13/S1  :       call POLLACK(TOP+1, MID, RIGHT)
.14     :       do PRINT_ENDIF
.15     :   ..endif
.15     :->(else)
.15     :   print error message "table entered in error"
.15     :   ..endif
        endsubroutine
```

On entry, the TOP condition row and LEFT and RIGHT rules which bound the subtable have been specified. The LEDT contains entries as described in Section 4.0.

This recursive subroutine processes the subtable of LEDT bounded by TOP and N (global) condition rows, and LEFT and RIGHT rule columns, and then prints the CRISP program for that subtable using the proper cosmetic nesting indicators.

On exit, the subtable program has been printed and the LEDT optimized.

Note: This subroutine is recursive in its description here; however, if the intended programming language does not support recursion, then an appropriate translation to iterative form using a stack may be substituted.

- .1 If LEFT and RIGHT coincide, only one action is possible, so
- .2 print the action text corresponding to LEFT, using TOP to compute the indentation level and cosmetics.
- .3 If TOP is not beyond N and more than one rule is in the table, then

Module Number S1
Module Name POILACK
Date 5/27/77

- .4 examine the rules to see if they all invoke the same action;
if so, set SAME true (otherwise, SAME is false).
- .5 When all actions are the same,
- .6 print that action, as in step .1, above.
- .7 Otherwise, the subtable requires refinement. Hence,
find the best condition B in this subtable according to the
cost criterion.
- .8 Exchange condition rows B and TOP between LEFT and
RIGHT
- .9 and collect all rules with Y in the top row occurring
between LEFT and RIGHT on the left. Set MID to the leftmost
rule with an N entry in the top row, or to RIGHT+1, if none
occurs.
- .10 Print "IF" followed by the condition text of the TOP
LEFT entry in the table. If this is not a positive
condition number, there is an error in the table, and will
eventually cause an error message at step .15 below:
Nevertheless, if negative, print "NOT" before the condition.
- .11 Perform the optimization of the "THEN" subtable and
print the "THEN" clause.
- .12 Separate with "(ELSE)" properly cosmetized using TOP to
compute nesting level. (Note: the scope of variable of a
do is the same as the module in which it appears.)
- .13 Perform optimization for the "ELSE" subtable, print the
"ELSE" clause,
- .14 and close with "ENDIF" cosmetized using TOP to compute
nesting level (see note in step .12, above).
- .15 The cases LEFT \neq RIGHT with TOP > C and LEFT > RIGHT
are errors caused by improper table entry.

Called from:

1.2
S1.11
S1.13

Module Number S1.12
Module Name PRINT_ELSE
Date 5/27/77

4. (S1.12) PRINT the ELSE keyword

TO PRINT_ELSE <*27 May 77*> MOD# S1.12
.1 print ":" and four spaces for TOP-1 times; then
follow by ":-> (ELSE)"
endto

On entry, the current value of TOP is the nesting level of the condition to which the ELSE corresponds.

This procedure prints the cosmetics and then "(ELSE)".

.1 If TOP were 1 then no cosmetics would precede the "IF", so the number of colon-plus-four-space repetitions is TOP-1.

Example L-2 489

Module Number S1.14
Module Name PRINT ENDIF
Date 5/27/77

4. (S1.14) PRINT the ENDIF keyword

To PRINT ENDIF <*27 May 77*> MOD# S1.14
.1 print ":" plus four spaces for TOP-1 times; then follow
by "...ENDIF"
endto

On entry, the current value of TOP is the nesting level of the condition to which the ENDIF corresponds.

This procedure prints the cosmetics and then "ENDIF".

.1 TOP-1 is the number of repetitions of the colon-plus-four-space field needed to reach the corresponding "IF".

490 *Appendix L*

Module Number S2
Module Name PRINT ACTION
Date 5/27/77

4. (S2) PRINT the ACTION text

Subroutine: PRINT ACTION (LEFT, TOP) <*27 May 77*> MOD# S2
.1 print ":" followed by 4 spaces, TOP-1 times; then
follow by the string ACTION(RULE(LEFT))
endsubroutine

On entry, the applicable rule has been determined to be the value of LEFT. RULE translates LEFT to an ACTION index. TOP indicates the nesting level of the IF structure to which this action applies.

This subroutine cosmetizes and prints the ACTION text for the LEFT RULE using TOP to determine indentation.

.1 LEFT is a column number, and RULE(LEFT) is the action index which applies. ACTION of this index is the string describing the action.

Called from:

S1.2
S1.6

Module Number S3
Module Name CHECK RULES
Date 5/31/77

4.(S3) Subroutine to CHECK RULES for identical actions

```
Subroutine: CHECK_RULE(LEFT, RIGHT, SAME) <*31 May 77*> MOD# S3
.1      Set SAME = true
.2      loop for i = LEFT by 1 while (i ≤ RIGHT AND SAME)
.3          ! if (RULE(i) ≠ RULE(LEFT))
.4              : set SAME = false
.5          ! :..endif
.5      !..repeat with next i
      endsubroutine
```

On entry, LEFT and RIGHT indicate the boundary of columns of a subtable. RULE contains the action indices for the subtable.

This subroutine examines each of the actions called for by the subtable.

On exit, if the subtable has only one action called for, SAME will be set to true; false otherwise.

- .1 Preset SAME to a true value to serve as a structure flag.
- .2-.5 Iterate through the subtable from LEFT to RIGHT until ended or until it has been found to have more than one action.
- .3 Test each rule against a fixed one (the LEFTmost),
- .4 and indicate failure when it occurs.

Called from:

S1.4

492 Appendix L

Module Number S4
Module Name EXCHANGE
CONDITIONS
Date 5/31/77

4. (S4) Subroutine to EXCHANGE CONDITION rows

Subroutine: EXCHANGE_CONDITIONS(TOP, B, LEFT, RIGHT) MOD# S4
 <*31 May 77*>
.1 loop for i = LEFT by 1 to RIGHT
.2 ! exchange LEDT(TOP, i) == LEDT(B, i)
.3 !...repeat with next i
 endsubroutine

On entry, TOP, LEFT, and RIGHT bound a subtable of LEDT, and B indicates the row chosen as best.

This subroutine interchanges the LEDT entries in the TOP and B rows between LEFT and RIGHT.

On exit, the LEDT subtable has the best row on TOP.

- .1-.3 From LEFT to RIGHT
.2 exchange row elements (the == operator signifies exchange).

Called from:

S1.8

Module Number S5
 Module Name PARTITION_RULES
 Date 5/31/77

4. (S5) Subroutine to PARTITION RULES

```

Subroutine: PARTITION_RULES(TOP, LEFT, RIGHT, MID)
              <*31 May 77*>                                MOD# S5
.1      let MID = LEFT, r = RIGHT
.2      loop while (MID ≤ r)
.3          if (LEDT(TOP, MID) > 0)
.4              let MID = MID+1
.5          :->(LEDT(TOP, r) < 0)
.6              let r = r-1
.7          :->(else)
.8              loop for i = TOP by 1 to N
.9                  exchange LEDT(i, MID) == LEDT(i, r)
.10             !..repeat with next i
.11             exchange PROB(MID) == PROB(r),
.12             RULE(MID) == RULE(r)
.13             let MID = MID+1, r = r-1
.14         !..endif
.15     !..repeat
.16     endsubroutine
  
```

On entry, the LEDT subtable bounded by TOP, LEFT, and RIGHT has the best row on TOP.

This subroutine exchanges columns until all true (positive) entries of the TOP row are on the left of false (negative) values.

On exit, the subtable has been partitioned, and MID contains the index of the first false (negative) TOP row entry (or RIGHT+1, if none exist).

- .1 This subroutine begins by presetting MID to the LEFT column and r to the RIGHT column of the current subtable,
- .2 and then iterates until all columns having true entries in the TOP row lie to the left of columns with false entries, as signalled by MID exceeding r.
- .3 During this iteration, if the TOP entry in the MID column is already true,
- .4 increment MID to the right, and repeat. This will continue until the MID column has a false TOP entry (or else until MID goes beyond r).
- .5 If column r now has a false entry,

494 Appendix L

Module Number S5
Module Name PARTITION_RULES
Date 5/31/77

- .6 decrement r to the left in search of a positive entry, until a true column r is found (or r goes beyond MID).
- .7 When MID and r columns have thus been found, iterate from the TOP of the subtable column to the bottom (row N). (Exchange from TOP to N is all that is needed since entries above TOP are identical in the two columns.)
- .8 Then exchange LEDT elements between columns MID and r.
- .9 Complete the column interchange by interchanging PROBABILITIES and RULES.
- .10 Advance MID and r to the next candidates for interchange, and repeat.

Called from:

S1.9

Module Number S6
Module Name PRINT_CONDITION
Date 5/31/77

3. (S6) Subroutine to PRINT the CONDITION string

```
Subroutine: PRINT_CONDITION(TOP, LEFT) <#31 May 77> MOD# S6
.1      if (LEDT(TOP, LEFT) > 0)
.2      :   print ":" followed by four spaces, TOP-1 times; then
        :   follow by "IF (" + CONDITION(LEDT(TOP, LEFT)) + ")"
        : ->(else)
.3      :   print ":" followed by four spaces, TOP-1 times; then
        :   follow by "IF (NOT" + CONDITION(-LEDT(TOP, LEFT)) + ")"
        :..endif
        endsubroutine
```

On entry, the value in the TOP LEFT position of LEDT is a + or - condition index into the CONDITION array. If there has been no table entry error, this index will be true (positive).

This subroutine normally, then, prints the corresponding CONDITION entry preceded by "IF (" and followed by ")", indented (using TOP) to the proper level. If an error has occurred and the index is negative, a "NOT" is inserted before the predicate. An error message will result later, a consequence of reaching S4.15.

On exit, the IF statement will always have been printed.

- .1 Check the index for a true value (positive).
- .2 If so, then print the predicate string in an IF statement. Plus indicates concatenation of strings.
- .3 If not, print it anyway, preceded by "NOT".

Called from:

S4.10

Module Number F1
 Module Name BEST
 Date 5/31/77

4. (F1) Function to pick the BEST row

Function: BEST(TOP, LEFT, RIGHT) <31 May 77*> MOD# F1

```

.1 let DELTA_T_MIN = infinity
.2 loop for c = TOP by 1 to N
.3   ! let DISCRIM = 1, DELTA_T = 0
.4   ! loop for rj = LEFT+1 by 1 to RIGHT
.5   !   ! loop for ri = LEFT by 1 to rj-1
.6   !   !   ! loop for k = TOP by 1 to c-1, c+1 to N
.7   !   !   !   ! let DISCRIM=DISCRIM*
.7   !   !   !   ! (LEDT(k,ri)=LEDT(k,rj))
.7   !   !   !   ! ..repeat with next k
.8   !   !   ! let DELTA_T=DELTA_T +
.8   !   !   ! (PROB(ri)*PROB(rj))*DISCRIM
.8   !   !   ! ..repeat with next ri
.8   !   !   ! ..repeat with next rj
.9   ! let DELTA_T = DELTA_T * TIME_COST(c)
.10  ! if (DELTA_T < DELTA_T_MIN)
.11  !   ! let DELTA_T_MIN = DELTA_T, B = c
.11  !   ! ..endif
.11  ! ..repeat with next c
.12  let BEST = B <returned value>
endfunction

```

On entry, TOP, LEFT and RIGHT bound the LEDT subtable for which the BEST row is sought.

This integer-valued function computes the extra expected times contributed to program execution by each condition in the subtable, and returns the value of the index of the row corresponding to the condition with least cost.

On exit, the LEDT is unaltered.

In this procedure, local variables correspond to values named in the cited reference text:

c	index of condition being considered; integer
ri,rj	rule indices r_i, r_j ; integers
DISCRIM	$D_c(r_i, r_j)$; real
DELTA_T	AT_c ; real
DELTA_T_MIN	minimum AT_c ; real

The integer local variable B in steps .11 and .12 records the current Best index candidate.

Module Number F1
Module Name BEST
Date 5/31/77

- .1 Preset the minimum cost metric to machine infinity, to permit lesser values to be considered.
- .2 Iterate through all conditions *c* left in the subtable.
- .3 For each new condition, preset the "don't care" DISCRIMinant to indicate a "don't care" is possible so far, and set the cost accumulator to zero.
- .4-.5 For each pair of rules *ri* and *rj* with $ri < rj$ in the subtable,
 - .6 and for all conditions $k \neq c$ in the subtable,
 - .7 compute the DISCRIMinant for rules *ri* and *rj* over all such conditions *k*. The relational expression in this equation takes a value 1 if true, 0 if false.
 - .8 Accumulate the probability-weighted DISCRIMinant into the cost function for condition *c*, over all rule pairs *ri*, *rj*.
- .9 Then weight by the TIME_COST for that decision, to give the final cost figure.
- .10 Compare this cost with the minimum cost found so far.
- .11 If less, replace the old value with the new, and record the condition index *c* as the Best candidate.
- .12 After all conditions have been checked, return *B* as the BEST condition.

5. REFERENCES

- .1 Tausworthe, Robert C., Standardized Development of Computer Software, Part I, Chapter 8, Prentice-Hall, Inc., Englewood Cliffs, NJ, 1977.
- .2 Pollack, S. L., "Conversion of LEDT's to Computer Programs," Commun. ACM, Vol. 8, No. 11, pp. 677-682, Nov. 1965.

END OF SPECIFICATION

EXAMPLE L-3

STANDARD RANDOM NUMBER GENERATOR FOR MBASIC™

1. INTRODUCTION

1.1 Purpose

This document presents and analyzes a machine-independent algorithm for generating pseudorandom numbers suitable for the standard MBASIC™ system. The algorithm used is the "polynomial congruential" or "linear recurrence modulo 2" method devised by the author in 1965. Numbers, formed as nonoverlapping, adjacent 28-bit words taken from the bit stream produced by the formula $a_{m+532} = a_{m+37} + a_m \pmod{2}$, will not repeat within the projected age of the solar system, will show no ensemble correlation, will exhibit uniform distribution of adjacent numbers up to 19 dimensions, and will not deviate from random runs-up and runs-down behavior.

1.2 Scope

The specifications herein contained address only machine-independent aspects of implementation. Standards and conventions, system environment and interfaces, test and verification details and other lower-level considerations are not covered.

1.3 General Description

The first MBASIC™ random number generator (Reference 6.1), implemented on the Univac 1108, used a linear congruential method (Reference 6.2), $x_{n+1} = 5^{15}x_n \pmod{2^{35}}$, followed by normalization to the range (0,1). This generator was used probably because it was already available in the U1108 statistics package. Empirical tests by users, however, later proved that the generator possessed very nonrandom correlation properties, indeed, especially if great care were not taken in specifying the initial "start" value.

This document describes an alternate generator of a type whose randomness has been theoretically shown to be vastly superior and which can be implemented on any computer, despite word length restrictions (the U1108 algorithm was tailored to 36-bit words). It is a machine-independent algorithm.

The generation method is almost as fast as the linear congruential method, but not quite. The ratio of speeds is about 3:1.

In 1965, the author (Reference 6.3) showed that numbers produced as successive binary words of length s taken L bits apart ($s \leq L$) from a linear (shift-register) bit-stream recursion of the form

$$a_n = c_1 a_{n-1} + \dots + c_{p-1} a_{n-p+1} + a_{n-p} \pmod{2}$$

form a pseudorandom sequence whenever the polynomial $f(x) = x^p + c_{p-1}x^{p-1} + \dots + c_1x + 1$ is primitive over $GF(2)$. The algebraic structure of these pseudorandom numbers provided a way of proving that, over randomly chosen starting values a_0, \dots, a_{p-1} in the numeric sequence, the correlation between numbers is essentially zero, being ostensibly equal to -2^{-p} , for all numbers in the sequence separated by less than $(2^p - s - 1)/L$, subject to the restriction $(L, 2^p - 1) = 1$. The author further showed that adjacent k -tuples of such numbers were uniformly distributed for $1 \leq k \leq (p/L)$.

The algorithm for computing the numbers is simple, especially when $f(x)$ is a primitive trinomial, say, $x^p + x^q + 1$, where $q < p/2$. An even greater simplification is possible, and used in this generator, when p is an even multiple of L , as is the case for the trinomial $x^{532} + x^{37} + 1$ (from Zierler and Brillhard, Reference 6.4). The generator based on this polynomial has period 1.4×10^{160} , has virtually no (average) correlation between any numbers separated by distances less than 5×10^{158} , has 28-bit precision numbers available, and has adjacencies up to 19 dimensions uniformly distributed. Runs-up and runs-down statistics up to length 16 are impeccable. The period and maximum correlation distance are, in fact, so great that the generator would have to produce numbers at a nano-second rate for more than 10^{142} years before nonrandom distribution or correlation effects would be noticeable as nonrandom. Almost 4×10^{14} numbers would have to be examined to detect deviations in runs-up and runs-down statistics as nonrandom.

These pseudorandom number generators have been widely studied (References 6.5-6.8) since 1965, both theoretically and empirically, and have been "promoted to pride of place in the field of pseudorandom number generation (Reference 6.7)."

Tootill (Reference 6.8) has even discovered generators of this type for which "there can exist no purely empirical tests of the sequence as it stands capable of distinguishing between it and [truly random sequences]." For reasons having to do with computer storage and precision, the generator of this article is, unfortunately, not one of these. Nevertheless, the generator

described is vastly superior to any linear congruential generator in existence.

The trinomial $x^{532} + x^{37} + 1$ (having $p = 532$, $q = 37$) has several things to recommend it: (1) 532 is factorable into 28 times 19, which means that 19 words each having 28 bits precision can be generated all at once by the algorithm; (2) up to and including 19-tuples of adjacently produced random numbers will be uniformly distributed; (3) 28 and 19 are both relatively prime to the period ($2^{532} - 1$), so no ill effects occur as a result of beginning new words every 28 bits; (4) the period and correlation distance are so great as never to be witnessed in the lifetime of the universe; and (5) the $q = 37$ value produces good runs-up and runs-down statistics (Reference 6.8), up to runs of length 16.

2. THE METHOD

The algorithm for producing the succeeding 532 bits from the current set of 532 bits in the generator is:

- (1) Left-shift the current 532-bit string by 37 bits, inserting 37 zeros on right, dropping 37 bits on the left.
- (2) Add modulo 2 (exclusive-or), the original and shifted 532 bits.
- (3) Right-shift this result by $532 - 37 = 495$ bits, supplying 495 zeros on the left, and dropping 495 bits on the right.
- (4) Add the results of (2) and (3), above, modulo 2, to give the next 532 bits.

The proof that this algorithm works is very simple, and the reader is invited to apply the algorithm to the bit string $a_1 a_2 \dots a_p$ and use the reduction formula $a_{p+m} = a_{q+m} + a_m$ (try it with $a_1 a_2 a_3 a_4 a_5$ with $a_{5+m} = a_{2+m} + a_m$ to see what is happening).

The RANDOMIZE function that initializes the generator uses a multiplicative linear congruential algorithm to generate the first 19-number "seed," from which the rest of the generated numbers grow. The particular value for the multiplier a in the algorithm

$$w_{n+1} = aw_n \pmod{2^L}$$

was chosen as 41,475,557 for the $L = 28$ case from theoretical results published in Ahrens and Dieter (Reference 6.9).

3. FUNCTIONAL SPECIFICATIONS

The random number generator will consist of one subroutine, RANDOMIZE(*starter*), and a parameterless function RANDOM. The *starter* parameter is a real number. If the *starter* is zero, the initial word w_0 is set

to 41,475,557 (the linear congruential multiplier); if *starter* is greater than zero, it is converted to its nearest integer equivalent, which then becomes w_0 ; if *starter* is less than zero, the computer clocktime is read and used as w_0 , a more or less random and unrepeatable starting value for the generator. RANDOMIZE then generates 18 more integer random words, w_1, \dots, w_{18} using the linear congruential method discussed in Section 2, above.

The RANDOM function returns a new real value at each invocation, the first 19 of which are generated by RANDOMIZE above, and the remainder using the linear recurrence algorithm. Integer values used within RANDOMIZE and RANDOM retain 28 bits precision. Values returned by RANDOM convert these to real numbers in the range (0,1).

On converting 28-bit fixed-point mantissas to real numbers, 8-digit precision results. Some machines, such as the Univac 1108, may have to reduce this precision in order to fit the floating-point exponent field into the word (the U1108 has only a 27-bit mantissa precision). In such cases, the most significant bits of the generated words shall always be retained so that all implementations produce essentially identical results, within machine precision. This philosophy is present in the algorithm that follows in Section 4.

Two values of the RANDOMIZE *starter* that round to the same internal fixed-point representation will produce the same random sequence; conversely, every unique fixed-point representation of the argument produces a unique random sequence. In addition, so long as the value of the argument is the same and stays within the precisions of two differing machines, the sequences produced on each will be the same, within machine precision.

4. PROGRAMMING SPECIFICATION

4.0 Overview

Managing the 532-bit shift-register is the main trick in implementing the method. The algorithm given in this section utilizes an array w_i , $i = 0, \dots, 18$ of b -bit computer words ($b \geq 28$) sufficient to encompass the 532 span of bits to be operated on (and retained), and delivered in 28-bit chunks whenever the RANDOM function is invoked.

On machines having word sizes smaller than 28 bits, double (or multiple) words will have to be used for each word in the algorithm below. XOR in the RANDOM function below signifies an "exclusive-OR" of the operand binary words.

Module Number 1
Module Name RANDOMIZE
Date 6/15/77

4.(1) The RANDOMIZE Subroutine

Subroutine: RANDOMIZE(starter:real)

<* This function declares and initializes a 19-element
<* array, w[0] , . . . , w[18] with random numbers generated
<* by a linear congruential method. An integer I is
<* set to zero to enable RANDOM to select w[0] as the
<* first random number. I and w are permanent data
<* structures, accessed only by RANDOMIZE and RANDOM.

```
.1      constant multiplier:integer = 41475557
.2      variable j:integer,I:integer,
      w:array [0..18] of universal integer
.3      if (starter<0)
.4      :      start:=clocktime<*returns current time of
      :      <*day as integer*>
.5      :-> (starter=0)
.6      :      start:=multiplier
      :-> (else) <*starter>0*>
.7      :      start:=fix(starter) <*floating-to-integer
      :      <*conversion*>
      :..endif
.8      w[0]:=start
.9      loop for j = 1 to 18
.10     | w[j] = (w[j-1]*multiplier) modulo 2**28
.11     |..repeat with next j
.12     I = 0 <*set up to pick w[0] as first random number*>
      endsubroutine
```


Module Number	2
Module Name	RANDOM
Date	6/15/77

4.(2) The RANDOM Function

Function: RANDOM:real

```

<* This algorithm makes use of a 19-word array,
<* w[0], ..., w[18], each with b  $\geq$  28 bits.
<* Each word contains precisely
<* 28 bits of the generator, right justified. A local
<* integer variable I on entry contains the index of
<* the word next to be returned as the random value.
<* Both w and I are permanent data structures, accessed
<* only by RANDOM and RANDOMIZE. The latter of these
<* initializes I to zero and w to the seed.

```

```

.1  variable j:integer
.2  if (I=19) <*all words have been used up*>
.3  :   I := 0 <*reset to first element in array*>
.4  :   loop for j=0 to 16 <*exclude last two words*>
.5  :       ! load w[j+1], w[j+2] into registers A0, A1
.6  :       ! left shift A1 by b-28 <*join bits in stream*>
.7  :       ! left shift A0,A1 by 9 <*q=37 is 28+9*>
.8  :       ! w[j] := (w[j] XOR A0) <*the recursion
.9  :       ! <*formula*>
.9  :       !..repeat with next j
.10 :       load w[18], w[0] into A0,A1 <*now compute w[17]:*>
.11 :       left shift A1 by b-28 <*join w[18], w[0]
.12 :       <*bit streams*>
.12 :       left shift A0,A1 by 9
.13 :       <*A0 now has final 19 bits of w[18]*>
.13 :       w[17] := w[17] XOR A0
.14 :       <*and first 9 bits of stream shifted 495*>
.14 :       load w[0], w[1] into A0,A1 <*do similarly
.15 :       <*for w[18]*>
.15 :       left shift A1 by b-28
.16 :       left shift A0, A1 by 9
.17 :       w[18] := (w[18] XOR A0)
.17 :   ...endif
.18 RANDOM = float(w[I])/2**28 <*convert to real*>
.19 I := I+1
endfunction

```

5. ANALYSIS AND EVALUATION

Note in the method that the number of computations required for generating the next p bits grows at most linearly in p . Assuming $p \gg L$, then partitioning the p generator bits into words of length L produces a number of words that also increases in proportion to p . Therefore, the number of computations for L -bit precision random numbers, to first-order effects, is independent of the recursion degree p . Making p large, however, has advantages in increasing randomness properties.

It is true that, as p increases, more and more registers are required to compute each new set of p bits, and shifting many registers at once presents a small inconvenience in most computer languages. These factors place a small speed and storage overhead on the generation process; but even this is not extreme due to the particular trinomial chosen.

Counting the number of elemental operations (load, store shift, etc.) for the algorithm, one finds about $10 + f$ operations per number generated, where f is the number of operations in "floating" the fixed-point number. The linear congruential algorithm requires only about $3 + f$ such operations, so the ratio of speeds is less than 3:1.

The RANDOM function is about $23 + f$ operations long, as compared to $3 + f$ for its linear congruential form, and data storage is 21 words versus 2. However, even though the program requires perhaps 7 times as much storage as the linear congruential form, the total is still probably under 50 locations, of negligible concern in most installations. The asymptotically random sequence of Tootill (Reference 6.8) requires 607 words to store the w-array alone. (This, coupled with the fact that only 23-bit precision was available in that generator, is why it was not considered here.)

The algorithm given has been implemented as the RANDOM function in the MBASICTM processor currently on the Caltech PDP-10 and the JPL Univac 1108 computers. All tests run on it so far validate the randomness properties claimed by the theory. In that theory, by the way, the only factor left to chance is the specification of the initial "seed." The stated uniformity, zero-correlation, and runs statistics are all based on the single assumption that the seed be chosen randomly. Of course, the default value canned in was not randomly chosen, but chosen specifically to look random except for the first word and, certainly, to the extent of the tests run, this appears to have worked beautifully.

It was also demonstrated that the generator is also capable (as is every known random number generator) of producing numbers with 3-sigma variations from randomness over a few thousand samples when the wrong seed is supplied.

As a coding check on the algorithm, the following 40 numbers suffice to establish probable correctness:

.1545085	→.6206040	→.2185256	→.7161612
.6887654	.6988667	.3544693	.9263908
.7998630	.3691845	.6903309	.9220639
.1908013	.8255893	.0641026	.9966978
.7919956	.4282176	.3321068	.6469416
.8004964	.5929163	.6594358	.9405825
.9601586	.8871027	.6285133	.6677605
.1921166	.3302294	.4858374	.8933217
.9015029	.2870978	.2324675	.3779590
.5168044	.5091386	.3086822	.2973689

The first 19 of these are the generated "seed," and the remaining 21 are the results of combining the stored numbers according to the method given.

6. REFERENCES

- 1 MBASIC™, Vol. I: *Fundamentals*, Jet Propulsion Laboratory, Pasadena, CA, p. 188, Aug. 1975.
- 2 Knuth, D. E., *The Art of Computer Programming, Vol. 2: Seminumerical Algorithms*, Addison-Wesley Publishing Co., Reading, MA, 1969.
- 3 Tausworthe, Robert C., "Random Numbers Generated by Linear Recurrence Modulo Two," *Math. Comp.*, Vol. XIX, No. 90, pp. 201-208, Apr. 1965.
- 4 Zierler, N., and Brillhart, J., "On Primitive Trinomials (Mod 2), II," *Inform. Contr.*, Vol. 14, No. 6, pp. 566-569, June 1969.
- 5 Whittlesey, J. R. B., "A Comparison of the Correlational Behavior of Random Number Generators for the IBM 360," *Comm. ACM*, Vol. 11, No. 9, pp. 641-644, Sept. 1968.
- 6 Neuman, F., and Martin, C. F., "The Autocorrelation Structure of Tausworthe Pseudorandom Number Generators," *IEEE Trans. Comp.*, Vol. C-25, No. 5, pp. 460-464, May 1976.
- 7 Tootill, J. P. R., *et al.*, "The Runs-Up and -Down Performance of Tausworthe Pseudorandom Number Generators," *ACM J.*, Vol. 18, No. 3, pp. 381-399, July 1971.
- 8 Tootill, J. P. R., *et al.*, "An Asymptotically Random Tausworthe Sequence," *ACM J.*, Vol. 20, No. 3, pp. 469-481, July 1973.
- 9 Ahrens, J. H., *et al.*, "Pseudorandom Numbers: A New Proposal for the Choice of Multipliers," *Computing*, No. 6, pp. 121-138, 1970.

END OF SPECIFICATION

EXAMPLE L-4

RANKING COMPETING FACTORS BY WEIGHTED DOMINANCES

1. INTRODUCTION

1.1 Purpose

This program rates a set of competing factors using weighted first- and second-order dominance scores derived from pairwise comparisons and weights input interactively by the user, and then prints a ranked list of the factors with scores.

1.2 Scope

The descriptions contained herein address only machine-independent aspects of implementation. Standards and conventions, environment and interfaces, test and verification details, and other lower-level considerations are not covered.

1.3 General Description

Given N factors that are to be ranked in order of importance, a pairwise judgment of the importance of one factor over another is called a *first-order dominance*. The number of times a particular factor dominates among the $N - 1$ other factors is the *first-order dominance score* for that factor.

A *second-order dominance* between two factors, say, i and k , is said to exist if there exists another factor j such that i dominates j , and j dominates k (in this case i exhibits a second-order dominance over k). Second-order dominance *strength* is defined as the number of such j that can be found for given i and k . The *second-order dominance score* for factor i is the sum of the strengths of all second-order dominances of i over other factors.

The above definitions make it impossible for a factor to exhibit either first- or second-order dominance over itself. Third- and higher-order dominances can similarly be defined and computed; however, they are of questionable utility, since they raise a philosophical problem, namely, that it is possible for a factor to exhibit a third- or higher-order dominance over itself. Although such calculations are straightforward, computation of third- and higher-order dominances are not included in the program capability described here.

First- and second-order scores can then be weighted and summed to compute weighted dominances for various purposes. The proper weighting factors for decision making are probably best determined by experimentation. Generally, first-order dominance scores provide a very reliable ranking, and second-order scores can be used to add weight to this ranking, or to break ties, or possibly to check the validity of the first-order results. If second-order dominances cause the rankings to change, the results may be suspect with regard to the rigor or true priorities that apply among competing factors.

An elementary discussion of dominance methods and their applications to behavioral science problems is given by Kemeny, J. G., Snell, J. L., and Thompson, G. L., in *Introduction to Finite Mathematics*, Second Edition, Prentice-Hall, Inc., 1966.

2. THE METHOD

Let N denote the number of factors to be ranked, and let First-order dominances be recorded in an $N \times N$ matrix F by setting $f_{i,k} = 1$ if factor i is preferred over factor k , and $f_{i,k} = 0$ otherwise. The sum of the entries in the i th row is then the first-order dominance score for factor i .

In the matrix $S = F^2$, the element $s_{i,k}$ will be equal to the number of times that a Second-order dominance of i over k exists, and is hence the strength of that dominance. The sum of the entries in the i th row of S is then the second-order dominance score for factor i .

If w_1 and w_2 are scalar weights input by the user and U is a $1 \times N$ all-ones vector, then the Resultant weighted score vector R is governed by the matrix equation

$$R = (w_1 F + w_2 S)U$$

3. FUNCTIONAL SPECIFICATIONS

The RANKING program operates in an interactive mode as presumed by this specification; suitable translation may be made to accommodate batch operation.

The user is prompted to input the number of factors to be ranked, followed by a request to input an equal number of short descriptive strings that identify these factors. Then, the program prompts the user to enter a judgment between pairwise alternatives for each pertinent pair of factors. The program names the factors to be judged before each judgment entry is made.

Example L-4 509

When all dominances have been registered, the program asks for weighting factors to be entered and, upon receiving these, prints the first-order, second-order, and weighted dominance scores.

The following example specifies the format of input and output dialogue. User entries are distinguished by the lighter type; the character " \leftarrow cr" denotes a carriage return. A ">" signals that a user entry is required.

ENTER THE NUMBER OF FACTORS TO BE RANKED: >4 \leftarrow cr

NOW NAME THESE 4 FACTORS:

>RELIABILITY \leftarrow cr

>SPEED \leftarrow cr

>STORAGE \leftarrow cr

>DOCUMENTATION \leftarrow cr

NOW RANK PAIRS AS FOLLOWS:

ENTER THE NUMBER (1 OR 2) OF FACTOR PREFERRED.

RELIABILITY OR SPEED?

>1 \leftarrow cr

RELIABILITY OR STORAGE?

>1 \leftarrow cr

RELIABILITY OR DOCUMENTATION?

>1 \leftarrow cr

SPEED OR STORAGE?

>1 \leftarrow cr

SPEED OR DOCUMENTATION?

>2 \leftarrow cr

STORAGE OR DOCUMENTATION?

>2 \leftarrow cr

NOW ENTER TWO WEIGHTS SEPARATED BY COMMA.

>1.0,0.5 \leftarrow cr

RANKING OF FACTORS IS AS FOLLOWS:

FACTOR	FIRST	SECOND	WEIGHTED
-----	-----	-----	-----
RELIABILITY	3	3	4.5
DOCUMENTATION	2	1	2.5
SPEED	1	0	1.0
STORAGE	0	0	0.0

DO YOU WANT TO TRY ANOTHER WEIGHTING? >NO \leftarrow cr

(program terminates)

The program shall accept up to 20 characters for each factor name. The number of factors and preferences must be integers. Weights are real numbers. The request for another weighting shall be answered either YES or NO.

If a YES is entered, the resulting program dialogue shall appear as above from the "NOW ENTER TWO WEIGHTS..." line on down.

Data type errors in user input are not checked by this program; recovery from such errors shall be configured, as can be best accommodated in the implementation language, so as to reprompt the user for correct entry.

4. PROGRAMMING SPECIFICATION

4.0 Overview

On execution, the program records the number of factors to be ranked in an integer variable *N* and the factor names in a string array *FACTOR*, which can hold *N* 20-character entries. Dominance values are recorded in the $N \times N$ matrix *FIRST*; squaring *FIRST* yields *SECOND*. Three $N \times 1$ arrays *ACCUM1*, *ACCUM2*, and *ACCUMW* hold evaluations of first-order, second-order, and weighted dominances, respectively.

Another $N \times 1$ array, *INDEX*, holds the indices that match entries in the *ACCUMW* array to the *FACTOR* names and *ACCUM1* and *ACCUM2* values. Then the *ACCUM1* and *INDEX* arrays are sorted in decreasing order using *ACCUMW* as a key. Printing of names and dominances proceeds in this sorted order (*ACCUM1*, *ACCUM2*, and *FACTOR* are addressed via *INDEX* in this printing).

Note: In programming languages that permit record data structures, then an array of *N* records with *FACTOR*, *ACCUM1*, *ACCUM2*, and *ACCUM* fields on each record can be sorted on the *ACCUMW* field, in which case, *INDEX* is not required.

Module Number	1
Module Name	RANKING
Date	6/15/77

4.(1) The RANKING Program

```

Program:  RANKING
.1      prompt_and_input_N
.2      declare_storage_arrays
.3      enter_factor_names
.4      enter_dominances
.5      compute_first_and_second_order_dominance_scores
.6      loop
.7          compute_weighted_dominances
.8          print_report
.9          prompt_for_another_weighting_and_accept_ANSWER
.10     repeat if (ANSWER = 'YES')
        endprogram

```

- .1 Declare an integer variable N, print the message "ENTER THE NUMBER OF FACTORS TO BE RANKED: >", accept the value of N, and check to assure $N > 2$ (the highest order of dominance). Reprompt and reaccept N if $N \leq 2$ was entered.
- .2 Declare the rest of the data structures used in the program. In languages that permit dynamic dimensioning of arrays, these arrays can be dimensioned to N in size. Otherwise, an array dimension larger than the envisioned range of N must be chosen. The declarations are: FACTOR[N]: string of 20 characters; INDEX[N], ACCUM1[N], ACCUM2[N], FIRST[N,N], SECOND[N,N] are integer arrays; ACCUMW[N,N] is a real array; i and j denote integer indices for these arrays; W1 and W2 are real variables; and ANSWER is a string variable capable of holding up to three characters. Initialize INDEX[i] = i for i = 1 to N, so as to record index-to-position information for later sorting.
- .3 Prompt with the message "NOW NAME THESE " + STR(N) + " FACTORS:", where STR is an integer-to-string conversion, and accept entry of N strings from the user into the FACTOR name arrays prompting by ">" before each. "+" above is string concatenation.
- .4 Print the message "NOW RANK PAIRS AS FOLLOWS:" followed by "ENTER THE NUMBER (1 OR 2) OF FACTOR PREFERRED." Then for each pair of indices i, j with $i > j$, prompt with the message FACTOR[i] + " OR " + FACTOR[j] + "?", then a carriage return and ">", and accept 1 or 2 (reprompt if entry is not 1 or 2). Set FIRST[i, j] and FIRST[j, i] to 0 and 1 to record dominance properly.

Module Number	1
Module Name	RANKING
Date	6/15/77

- .5 Square the matrix FIRST and store result in SECOND. Sum the rows of each and insert results in ACCUM1 and ACCUM2 arrays, respectively.
- .6 Then, to accommodate several weighting factors, if desired,
- .7 prompt user with the message "NOW ENTER TWO WEIGHTS SEPARATED BY COMMA." followed by a carriage return and " >". Then accept real weights W1 and W2, compute weighted dominances, and store these in ACCUMW.
- .8 Sort the ACCUMW array in descending order of value from ACCUMW[1] to ACCUMW[N]. Use the Bubble-sort technique (cf. Example 7.3.3.1 in Part I of this text, but replace ">" by "<" in the "ELEMENTS...OUT OF ORDER" macro). Then print the report header shown in Section 3, followed by FACTOR [INDEX [i]], ACCUM [INDEX [i]], ACCUM2 [INDEX [i]], and ACCUMW [i] for i = 1 to N.
- .9 Print the message "DO YOU WANT TO TRY ANOTHER WEIGHTING? >", and accept ANSWER as "YES" or "NO". Reprompt if anything else is entered.
- .10 Repeat from step 6 if another weighting was desired; otherwise terminate execution.

END OF SPECIFICATION

APPENDIX M

USEFUL STANDARD FORMS

The 23 forms contained in this final appendix are typical among those commonly in use in medium- and larger-scale software projects. These are forms for project planning and management, functional design, status reporting, configuration management, and quality assurance. The use of such materials constitutes what some have termed a "forms approach" to software development, which mirrors practices used in almost every successful engineering and business endeavor. Depending on the size and intended lifecycle of the software, the need for the information such forms contain is real, not the mere whim of an administrator whose lust tends toward rigid formal measures and banal routine procedures. Properly done, the use of forms in software development tends to standardize interfaces among people and across organizational lines, and can, at the same time, serve to turn "paper work" into "working paper." Most of the forms shown in this appendix can most assuredly be computerized, both in the process of collecting the data required to fill them out, as well as in the processing, output, and distribution of those data to their intended ends.

However, whether such forms as these are eventually computerized or not for project usage will depend largely on how each of the selected forms seems to interface most effectively with the conglomerate of human activities for which it is intended. Forms are viewed as media of expression in the design process, of working documentation during software construction and test, of communication among team members, of status monitoring and project control during implementation, and of historical value for the future. New forms may be added with the same goals.

The forms here are presented without explanation as to their exact recommended usage, the specific meaning of the blanks requiring entry by the user, the criteria for use or interpretation, standards for nomenclature

and coding of identifiers, or the procedures to be followed to make the forms effective. I do not mean to imply by this omission that I believe the forms are self-explanatory (even though strived for), not needing such accompanying documentation for fully effective and non-ambiguous use. Nor do I contend that the set of forms given is a complete and sufficient set of such materials needed by a project. Some projects may need more, some less.

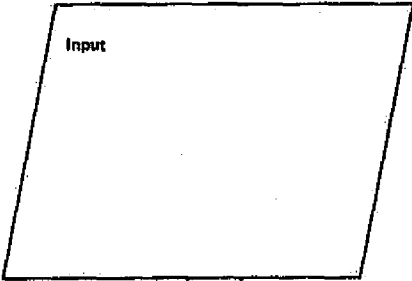

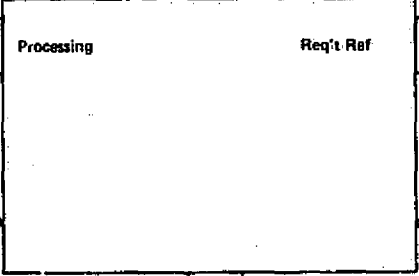

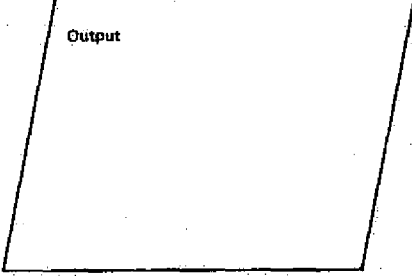




Rather, the forms represent examples upon which project, organization, or industry standards are or can be built. They further illustrate the types of information required by programmers, managers, QA personnel, and others associated with the software being developed in coping with their own various and diversified tasks.

The forms of this appendix find their origins, both in format and content, in several JPL, Military/DOD, and industry sources. I have extended some of these forms, simplified others, and merely translated or rearranged still others, so that all of the exhibited forms bear a hoped-for family resemblance.

Work Breakdown Structure Detailed Task Description			
Title:		Task:	
		Identifier:	
		Date:	
Task Mgr:	Phone:	Duration:	to
Task Description			
Time-Phased Manpower	days	function	support
name			
Task Scope (estimated)	Text	Flowcharts	Figures Code
Task Needs			
Task Deliverables			
Precedent tasks		Interfacing tasks	

516 Appendix M

Decision Table																					
Description:										Title:											
Prepared by:										Phone:				Identifier:				Date:			
Conditions		Rule	1	2	3	4	5	6	7	8	9	10	11	12	13	14	ELSE	COST			
Prob																					
Actions		Rule	1	2	3	4	5	6	7	8	9	10	11	12	13	14	ELSE				
Sequential Test Procedure																					
Signature							Date			Signature							Date				

Input – Processing – Output Table										
Description: 	Mode: 									
Prepared by: 	Phone: 	Identifier: 								
<div style="display: flex; justify-content: space-around; align-items: center;"> <div style="text-align: center;"> <p>Input</p>  </div> <div style="text-align: center;">  </div> <div style="text-align: center;"> <p>Processing</p>  </div> <div style="text-align: center;">  </div> <div style="text-align: center;"> <p>Output</p>  </div> </div> <div style="display: flex; justify-content: space-between; margin-top: 10px;"> <div style="width: 30%;"> <p>Software Interfaces</p>   </div> <div style="width: 30%; text-align: center;"> <p>Req't Ref</p> </div> <div style="width: 30%; text-align: right;"> <p>System/Subsystem Interfaces</p>   </div> </div>										
<div style="display: flex; justify-content: flex-end; align-items: flex-start;"> <table border="1" style="border-collapse: collapse;"> <thead> <tr> <th style="padding: 2px 5px;">Signature</th> <th style="padding: 2px 5px;">Date</th> </tr> </thead> <tbody> <tr><td style="height: 20px;"></td><td></td></tr> <tr><td style="height: 20px;"></td><td></td></tr> <tr><td style="height: 20px;"></td><td></td></tr> </tbody> </table> </div>			Signature	Date						
Signature	Date									

Software Technical Progress Report									
Title:					Task:				
Task Mgr:					Phone:				
Status Summary					Identifier:				
Problems					Date:				
Forecast					Period: to				
					% Complete: _____				
					Effort to complete: _____				
					Remaining Resources: _____				
					Schedule Impact: _____				
Resources:									
Name				Function				Effort/other expenditure	
Accomplishments									
<div style="display: flex; flex-direction: column; gap: 5px;"> <div>text</div> <div>flowcharts</div> <div>figures</div> <div>tables</div> <div>code</div> <div>test</div> </div>	this report		cumulative		final (est)		percent (est)		
	units	effort	units	effort	units	effort	units	effort	
Scheduled completion date:		Prepared by:				Date:			

Software Anomaly Report						
Software Item:				Anomaly No.:		
Initiated by:				Phone:		Date:
Version	Config/OS	Cat-Prior	Detect. Date	Response Date	Log Date	
References						
Problem Description (symptoms and conditions of occurrence)						
Data	Test Procedure		Test Version			
Attachment: Yes <input type="checkbox"/> No <input type="checkbox"/>						
Suspected Cause, Comments, and Recommendation						

Software Anomaly Correction Report						
Software Item:				Anomaly No.:		
Closed by:				SW ID:		Date:
Phone:						
Corr Vers	Config/OS	Control date	Retest Date	Master Tape	Backup Tape	
Correction Summary						
Retest Data		Retest Procedure		Retest Version		
Attachment: Yes <input type="checkbox"/> No <input type="checkbox"/>						
Affected Items						
Problem Category:		Test procedure <input type="checkbox"/> Hardware <input type="checkbox"/> User manual <input type="checkbox"/> Functional spec <input type="checkbox"/> Design error <input type="checkbox"/> Ops system <input type="checkbox"/> Ops manual <input type="checkbox"/> Programming spec <input type="checkbox"/> Coding error <input type="checkbox"/> Human error <input type="checkbox"/> _____				

Software Anomaly Status Report

[illegible]

Software Change Request						
Software Item:				Req. No.:		
				SW ID:		
Submitted By:				Date:		
				Attachment: Yes <input type="checkbox"/> No <input type="checkbox"/>		
Phone:						
Version	Config/OS	Reply Date	Need Date	Anomaly No.	Cat. Prior	
Change Description						
Justification						
Other Items Affected						
Action	Disapproved <input type="checkbox"/>	Grounds _____				
	Analysis recommended <input type="checkbox"/>					
	Approved <input type="checkbox"/>	Proviso _____				
Comments						
Signature		Date	Signature		Date	

Software Change Analysis Report						
Software Item:				Req. No.:		
				SW ID:		
Prepared by:				Date:		
				Attachment: Yes <input type="checkbox"/> No <input type="checkbox"/>		
Version	Config/OS	Temporary <input type="checkbox"/> Permanent <input type="checkbox"/>	Lifetime (yrs)	Anomaly No.	Cat - Prior	
Analysis Summary						
<div> <div>man-hrs _____</div> <div>comptr hrs _____</div> <div>doc hrs _____</div> <div>other _____</div> <div>total _____</div> </div> <div> <div>\$ _____</div> <div>\$ _____</div> <div>\$ _____</div> <div>\$ _____</div> <div>\$ _____</div> </div>						
Items Affected						
Recommended Action/Alternatives						
<div> <div>Action</div> <div>Disapproved <input type="checkbox"/></div> <div>Approved <input type="checkbox"/></div> <div>Comments</div> </div> <div> <div>Grounds _____</div> <div>Proviso _____</div> </div>						
Signature		Date		Signature		Date

Software Change Order					
Software Item:			C.O. No.:		
			SW ID:		
			Date:		
Assigned to:		Phone:		Attachment: Yes <input type="checkbox"/> No <input type="checkbox"/>	
Version	Config/OS	Temporary <input type="checkbox"/> Permanent <input type="checkbox"/>	Due Date	Remove Date	Cat-Prior
Work Description					
Task	start date	compl date	manpower cost	equipment cost	account codes
Design					
Procurement					
Hardware fab/mod.					
Software coding					
Documentation					
Testing					
Training					
Mod kits					
Spares					
Other					
Totals					
Signature		Date	Signature		Date

Specification Change Notice

Software Item:		SW ID:	
Issued by:		Date:	
		Version	Config/OS
Phone:			

Recipients of this notice are hereby informed that the specifications referenced above have been changed. Descriptions of the changes are logged below and materials for effecting said changes are furnished herewith.

Item No.	SSD Section	Pages	Instructions	Change Order No.	Effective Date

Software Audit Report					
Software Item:				SW ID:	
				Date:	
Audited by:		Phone:			
Version	Config/OS	Control Date	Audit Date	Audit Type	Audit Method
References					
Items Audited					
Discrepancies/Exceptions					
Status Summary					
Signature		Date		Signature	
				Date	

Action Item Log

[illegible]

Software Configuration Log

Software Item:

Log.No.:

SW ID:

Logged by:

Phone:

Page:

[illegible]

Configuration Status Report

[illegible]

Software Standard Waiver			
Software Item:		Waiver No.:	
Requested by:		SW ID:	
Phone:		Date:	
Description of Standard to be Waived			
Scope of Waiver (items to which waiver applies)			
Justification and Analysis			
Proposed Alternate to Standard			
Action		Grounds	
Comments		Proviso	
Signature	Date	Signature	Date

Project:		Task:	
		Identifier:	
		Date:	

MILESTONES			
1			
2			
3			
4			
5			
6			
7			
8			
9			
10			
11			
12			
13			
14			
15			
16			
17			
18			
19			
20			
21			

Legend (darken when accomplished): <div style="display: flex; justify-content: space-between; align-items: flex-start;"> <div style="width: 45%;"> <div style="display: flex; align-items: center; margin-bottom: 5px;"> <div style="width: 15px; height: 15px; border: 1px solid black; border-radius: 50%; margin-right: 5px;"></div> Input due </div> <div style="display: flex; align-items: center; margin-bottom: 5px;"> <div style="width: 15px; height: 15px; border: 1px solid black; margin-right: 5px;"></div> Original-milestone </div> </div> <div style="width: 45%;"> <div style="display: flex; align-items: center; margin-bottom: 5px;"> <div style="width: 15px; height: 15px; border: 1px solid black; transform: rotate(45deg); margin-right: 5px;"></div> Rescheduled milestone </div> <div style="display: flex; align-items: center;"> <div style="width: 15px; height: 15px; border: 1px solid black; transform: rotate(45deg); margin-right: 5px;"></div> Delivered Item </div> </div> </div>		<div style="display: flex; justify-content: space-between; align-items: flex-start;"> <div style="width: 45%;"> <div style="display: flex; align-items: center; margin-bottom: 5px;"> <div style="width: 15px; height: 15px; border: 1px solid black; background-color: #cccccc; margin-right: 5px;"></div> Work force </div> <div style="display: flex; align-items: center;"> <div style="width: 15px; height: 15px; border: 1px solid black; background-color: #cccccc; margin-right: 5px;"></div> Work planned </div> </div> <div style="width: 45%;"> <div style="display: flex; align-items: center; margin-bottom: 5px;"> <div style="width: 15px; height: 15px; border: 1px solid black; background-color: #cccccc; margin-right: 5px;"></div> Rescheduled work </div> </div> </div>
PREPARED BY:	DATE:	
APPROVED BY:	DATE:	

Project:												Task:			
												Identifier:			
												Date:			
MILESTONES															
1															
2															
3															
4															
5															
6															
7															
8															
9															
10															
11															
12															
13															
14															
15															
16															
17															
18															
19															
20															
21															
Legend (darken when accomplished):															
<input type="checkbox"/> Input due <input type="checkbox"/> Rescheduled milestone <input type="checkbox"/> Work force <input type="checkbox"/> Original milestone <input type="checkbox"/> Delivered Item <input type="checkbox"/> Work planned <input type="checkbox"/> Rescheduled work												PREPARED BY:		DATE:	
APPROVED BY:												DATE:			
1 YEAR SCHEDULE															

Project:																			Task:						
																			Identifier:						
																			Date:						
MILESTONES		J	J	M	A	M	J	J	A	S	O	N	D	J	J	M	A	M	J	J	A	S	O	N	D
1																									
2																									
3																									
4																									
5																									
6																									
7																									
8																									
9																									
10																									
11																									
12																									
13																									
14																									
15																									
16																									
17																									
18																									
19																									
20																									
21																									

Legend (darken when accomplished):

- ☐ Input due ▽ Rescheduled milestone
- Original milestone ◇ Delivered Item

n Work force

☐ Work planned ☐ Rescheduled work

PREPARED BY:

DATE:

APPROVED BY:

DATE:

Project:		Task: Identifier: Date:	
MILESTONES			
1			
2			
3			
4			
5			
6			
7			
8			
9			
10			
11			
12			
13			
14			
15			
16			
17			
18			
19			
20			
21			

Legend (darken when accomplished):

☐ Input due

☐ Original milestone

☐ Rescheduled milestone

☐ Delivered Item

Work force

Work planned

Rescheduled work

PREPARED BY:
APPROVED BY:

DATE:
DATE:

Project:	Task:
	Identifier:
	Date:
MILESTONES	J F M A M J J A S O N D J F M A M J J A S O N D J F M A M J J A S O N D J F M A M J J A S O N D J F M A M J J A S O N D
1	
2	
3	
4	
5	
6	
7	
8	
9	
10	
11	
12	
13	
14	
15	
16	
17	
18	
19	
20	
21	

Legend (darken when accomplished):

- ☐ Input due ▽ Rescheduled milestone
- ☐ Original milestone ◇ Delivered Item

n
 Work force
 Work planned
 Rescheduled work

PREPARED BY:

APPROVED BY:

DATE:

DATE:

REFERENCES

1. *Software Implementation Guidelines and Practices*, DSN Standard Practice 810-13, Jet Propulsion Laboratory, Pasadena, CA, Aug. 1975.
2. *Preparation of Software Requirements Documents*, DSN Standard Practice 810-16, Jet Propulsion Laboratory, Pasadena, CA, Dec. 1975.
3. *Preparation of Software Definition Documents*, DSN Standard Practice 810-17, Jet Propulsion Laboratory, Pasadena, CA, July 1976.
4. *Preparation of Software Specification Documents*, DSN Standard Practice 810-19, Jet Propulsion Laboratory, Pasadena, CA, Mar. 1977.
5. *Preparation of Software Operator's Manuals*, DSN Standard Practice 810-20, Jet Propulsion Laboratory, Pasadena, CA, Feb. 1977.
6. *Preparation of Software Transfer to Operations Documents*, DSN Standard Practice 810-21, Jet Propulsion Laboratory, Pasadena, CA, Nov. 1976.
7. Tausworthe, R. C., "Stochastic Models for Software Project Management," *Deep Space Network Progress Report 42-37*, pp. 118-126, Jet Propulsion Laboratory, Pasadena, CA, Feb. 1977.
8. Boehm, B. W., et al., *Software Development and Configuration Management Manual*, TRW Systems Group, Santa Monica, CA, Dec. 17, 1973.
9. Royce, W. W., "Managing the Development of Large Software Systems: Concepts and Techniques," Western Electronics Conference, Hollywood Park, CA, Aug. 1970.
10. Musa, J. D., and Woomey, F. N., Jr., "Software Project Management," *Bell System Technical Journal (Safeguard Supplement)*, pp. S245-S259, 1975.

540 *References*

11. *Fundamentals of MBASICTM*, Vols. 1 and 2, Jet Propulsion Laboratory, Pasadena, CA, Feb. 19, 1974.
12. Foster, R. A., *Introduction to Software Quality Assurance*, Space Systems Division, Lockheed Missiles and Space Co., Sunnyvale, CA, 1973.
13. Tate, K., "DSN Software Quality Assurance Statement of Work," Internal Memorandum, Jet Propulsion Laboratory, Pasadena, CA, Aug. 1973.
14. Buxton, J. N., *et al.*, *Software Engineering Techniques*, Report on a Conference sponsored by the NATO Science Committee, Rome, Italy, October 27-31, 1969 (available through Scientific Affairs Division, NATO, Brussels, 39, Belgium).
15. Wolverton, G., and Schick, R., "Assessment of Software Reliability," *Proc. 11th Meeting*, German Operations Research Society, Hamburg, Sept. 1972.
16. Musa, J. D., "A Theory of Software Reliability and Its Application," *IEEE Trans. Software Eng.*, Vol. SE-1, No. 3, pp. 312-327, Sept. 1975.
17. Jelinski, Z., and Moranda, P. B., "Software Reliability Research," Conference on Statistical Methods for the Evaluation of Computer Systems, Brown University, Providence, RI, Nov. 1971.
18. "Improvement Needed in Documenting Computer Systems" *Report to the Congress*, by the Comptroller General of the United States, Washington, DC, Oct. 8, 1974.
19. Gray, M., and Landon, K., *Documentation Standards*, Brandon/Systems Press, Inc., Princeton, NJ, 1969.
20. Weinberg, G., *The Psychology of Computer Programming*, D. Van Nostrand Reinhold Co., NJ, 1971.
21. NASA Guideline, *Computer Program Documentation Guidelines*, NHB 2411.1, National Aeronautics and Space Administration, Washington, DC, July 1971.
22. *JPL Drafting Manual*, Document JPL-STD00001A, Jet Propulsion Laboratory, Pasadena, CA, Sept. 15, 1969.
23. *Military Standard Engineering Drawing Practices*, MIL-STD-100A, Department of Defense, Washington, DC, Oct. 1, 1967.
24. Barry, B. S., and Naughton, J. J., "Chief Programmer Team Operations," *Structured Programming Series*, Vol. X, Rome Air Development Center, Griffiss Air Force Base, NY, Jan. 22, 1975.

25. Tinanoff, N., and Luppino, F. M., "Programming Support Library Program Specifications," *Structured Programming Series*, Vol. VI, RADC-TR-74-300, Rome Air Development Center, Griffiss Air Force Base, NY, Nov. 22, 1974.
26. Brinch-Hansen, P., *The Solo Operating System*, Information Science Department, California Institute of Technology, Pasadena, CA, July 1975.
27. Wegbreit, B., *Multiple Evaluators in an Extensible Programming System*, Harvard University Center for Research in Computing Technology, Cambridge, MA, Mar. 1973.
28. Luppino, F. M., and Smith, R. L., "Programming Support Library Functional Requirements," *Structured Programming Series*, Vol. V, RADC-TR-74-300, Rome Air Development Center, Griffiss Air Force Base, NY, July 24, 1974.
29. *Department of Defense Requirements for Higher-Order Languages*, Defense Advanced Research Projects Agency, Arlington, VA, Apr. 1, 1976.
30. Goodenough, J. B., and Shaffer, L. H., *A Study of High Level Languages*, Vols. I and II, U. S. Army Electronics Command, EOM-75-073-F KD/A-021-206, Fort Monmouth, NJ, Feb. 1976.
31. Enslow, P. H., et al., *Implementation Languages for Real-Time Systems; Part I—Its Implementation and Acceptance*, ERO-2-75, European Research Office, London, England, Apr. 1975 (NTIS No. AD/A-008 977, U. S. Department Commerce, Springfield, VA).
32. Enslow, P. H., et al., *Implementation Languages for Real-Time Systems; Part II—Language Design*, ERO-2-75, European Research Office, London, England, Apr. 1975 (NTIS No. AD/A-008 978, U. S. Department Commerce, Springfield, VA).
33. Nicholls, J. E., *The Structure and Design of Programming Languages*, Addison-Wesley Publishing Co., Inc., Reading, MA, 1975.
34. Knuth, D. E., *The Art of Computer Programming: Vol. 1, Fundamental Algorithms*, pp. 78-85, Addison-Wesley Publishing Co., Inc., Reading, MA, 1969.
35. Tennent, R. D., *A Proposed Generalization of Pascal*, Technical Report. No. 75-32, Department of Computing and Information Science, Queen's University, Kingston, Ontario, Canada, Feb. 1975.
36. Liskov, B. H., and Zilles, S. N., "Specification Techniques for Data Abstractions," *IEEE Transactions on Software Engineering*, Vol. SE-1, No. 1, pp. 7-18, Mar. 1975.

542 References

37. Hoare, C. A. R., "Data Reliability," *Proc. 1975 International Conference on Reliable Software*, Los Angeles, CA, Apr. 21-23, 1975.
38. AUTOFLOW Computer Documentation System, Applied Data Research, Inc., Princeton, NJ, June 1970.
39. Pardee, S., "The BELFLOW System," in *Automated Methods of Computer Program Documentation*, NASA-TM-X-66196, Goddard Space Flight Center, Greenbelt, MD, Nov. 1970.
40. Brinch-Hansen, P., *Concurrent Pascal Report*, Information Science Department, California Institute of Technology, Pasadena, CA, June 1975.
41. Lefkovits, H. C., et al., *A Progress Report on the Activities of the CODASYL End User Facility Task Group*, CODASYL End User Facility Task Group Report, P. O. Box 297, Harvard, MA, June 1975.
42. *American National Standard Vocabulary for Information Processing*, ANSI X3.12-1970, American National Standards Institute, Inc., New York, Feb. 18, 1970.
43. *Webster's New Intercollegiate Dictionary*, G. and C. Merriam Co., Springfield, Mass. 1961.
44. *Guidelines for Documentation of Computer Programs and Automated Data Systems*, Federal Information Processing Standards Publication, FIPS PUB #38, U. S. Department of Commerce, National Bureau of Standards, Feb. 15, 1976.
45. Waish, D., *A Guide for Software Documentation*, Advanced Computer Techniques Corp., New York, 1969.
46. *Guidelines for Preparation of Mark III Data System Software*, Deep Space Network Data Systems Development Section, Jet Propulsion Laboratory, Pasadena, CA, (internal working document).
47. Smith, R. L., "Management Data Collection and Reporting," Final Report, *Structured Programming Series*, Vol. IX, RADC-TR-74-300, Air Force Systems Command, Rome Air Development Center, Griffiss AFB, New York, and Army Computer Systems Command, Fort Belvoir, Va., Oct. 1974.
48. MBASICTM, Vol. II: *Appendices*, Jet Propulsion Laboratory, Pasadena, CA, Aug. 1975.

INDEX

NOTE: Page numbers that appear in boldface type indicate the reference for the primary discussion or definition of an item.

- Abnormal terminations, 53, 92
- Abort, 333
- Abstraction, 40
- Abstract data type, 196
- Acceptance testing, 104, 129, 137
- Access rights, 196
- Access time, 45
- Accuracy, 12
- Action entries (Decision Table), 31
- Action items (Decision Table), 84
- Algorithm, 24, 60, 79
 - table driven, 44
- Annotation, 70, 82, 195
- Anomaly discovery, repair, 112, 119, 124
- Anomaly reporting, 396
- ANSI, 27, 65, 205, 224, 237, 313, 321
- Arbitration, 56, 94
- Architectural design, 4, 7, 36, 45, 65, 220, 225, 263
- Archiving, 101, 112, 117
- As-built, 4, 58, 101
- ASCII, 329
- AT, 334, 350
- Audit, 60, 84, 103, 117, 132
 - code, 135
 - requirements, 159
- Automated tools, 171
- Backslash, 324
- Base language, 187, 200, 312
- Block diagram, 38
- CALL, 335, 355
- CALLX, 335, 355
- Cancel, 338
- Canonic structures, 53
- Cascaded testing, 128
- CASE, 336
- Change control, 33, 113, 141, 189
- Changes in priorities, schedules, etc., 13
- Chart, 40, 71
- Checkout, 103
- Chief Programmer Team, 84
- Clarity, 61
- Class A documentation, 58, 65, 158, 416
- Class B documentation, 159, 416
- Class C documentation, 80, 160
- Class D documentation, 80, 162
- Code auditing, 135
- Code module, 86
- Code submodule, 86
- Coding conventions, 95
- Coding standards, 85
- Collecting node, 70
- Comments, 202, 320, 321, 327, 331
- Common definition, 18
- Competing characteristics, 6

- Competing methods, 36
- Compile module, 86
- Compilers, 321
- Compile-time option, 90
- Complexity, 221
- Concurrent documentation, 77
- Concurrent processes, 54, 94, 107, 114
- Confidence measurement, 106, 405
- Configuration control, 71, 79, 137
- Configuration management, 137, 216, 513
- Connections, 16
- Consistency, 55, 107, 113, 186, 211
- Contingencies, 10
- Control copy, 98
- Control definitions, 18
- Control flags, 63, 104
- Control flow, 238
- Control logic, 27, 50, 58, 61, 103
- Control sublanguage, 187
- Control structures, 49
- Core-swapping, 90
- Correctness, 43, 93, 103, 114, 222
- Correctness assertions, 90
- Cosmetics, 330
- Cost and schedule drivers, 4
 - estimation, 13
- Cost bounds, 6
- Coupling modes, 39
- Cousin modules, 82
- CRISP, 40, 91, 187, 222, 309
 - FLOW, 81, 205, 310, 320
 - PDL, 24, 36, 78, 86, 134, 151, 190, 201, 310, 318
 - translators, 321
- Criteria, 2, 8, 15, 57, 317
- Critical path method, 417
- Critical task, 419
- Cross-reference, 65, 68, 83, 96, 247, 320, 321, 344, 365
- CYCLE, 345
- Data base, 177, 223, 260
 - interfaces, 389, 411
 - specification, 286
- Data, 222
 - flow, 151, 177, 238
 - interface, 34
 - packing, 45
 - structure, 39, 63, 77, 90, 194
 - structure definition table, 77, 134
 - type, 195
- Deadline, 55, 107
- Deadlock, 55, 107
- Decision box, 70
- Decision collecting nodes, 327
- Decision table, 24, 31, 32, 39
- Declaration, 195
- Default values, 58
- Definition standards, 1
- Deliverables, 256, 405
- Delivery readiness, 405
- Department of Defense, 196
- Design, 223
 - aids, 311
 - definition, 46
 - feasibility study, 7
 - phase, 46
 - philosophy, 87
 - practices, 57
 - tradeoff criteria, 36
 - verification, 118
- Detail class, 156
- Development support, 257
- Development testing, 103, 136
- Dewey-decimal, 20, 65, 79, 100, 134
- Diagnostic procedures, 113
- Discrepancy report, 112, 113, 119
- Display, 338
- DO, 335, 352
- Documentation, 19, 33, 58, 94, 111, 138, 147, 172
- Documentation levels, 156
 - Class A, 58, 65, 158, 416
 - Class B, 159, 416
 - Class C, 80, 160
 - Class D, 80, 162
- Concurrent, 77
- Format 1, 163
- Format 2, 164
- Format 3, 165
- Format 4, 165
- Document library, 174
- Dominances, 6, 507
- DSDT, 77
- Earliest finish, 419
- Earliest start, 419

- Editing, 210
- ELSE, 336, 338, 342
- ELSE-rule, 57
- ENABLE, 338
- Engineering, 224
- Environment, 16, 181, 282, 314
 - operational, 128
- Estimation, cost, schedule, 13
- Examples, 415
- Exit, 339, 351, 352
- External characteristics, 3, 15, 19
- External data bases, 37

- Fail safe, 19
- Fail soft, 19
- Fatal errors, 53
- Feasibility, 3, 45
- Field name, 45
- File naming, 101
- Finish, 340
- Flag stack, 52
- Float, 419
- Flowchart, 45, 65, 134, 151, 177, 201, 237, 311, 320
- Flowline, 105
- Fork, 239, 340
- FORK-JOIN, 94
- Format 1 documentation, 163
- Format 2 documentation, 164
- Format 3 documentation, 165
- Format 4 documentation, 165
- Format category, 156
- Format quality, 163
- Forms, 213, 513
- FORTRAN, 95, 195
- FRD (Functional Requirements Document), 225, 232, 251
- Function, 21, 39, 341, 353
- Functional correctness, 63
- Functional testing, 131
- Function calls, 82
- Fundamental scalar, 195

- Garbage collection, 45
- Glossary, 31, 34, 98, 134, 219, 326, 344, 365
- Goals, program, 13
- Grade, project, 7
- Graphics, 24, 174, 202

- Hierarchical design, 49
- Hierarchic input-processing-output (HIPO), 23, 225, 259
- Hierarchic refinement, 251
- Hierarchy, 225
- HIPO, 23, 225, 259
- Horizontal striping, 27
- Human factors, 13, 19, 148, 171, 176, 513

- Identifiers, 323
- IF, 342
- IF modifier, 343
- Imperative mood, 76, 82
- Indenting, 96
- Information, 20, 226
 - flow, 24, 27, 28
 - reporting, 396
- Initialization, 70
- In-line procedures, 90
- Input/output, 199
 - requirements, 64
 - sublanguage, 189, 200
 - type, 199
- Input-processing-output table, 29
- Integrity, 143
- Interactive, 184, 211, 191
- Interfaces, 16, 49, 61, 104, 109, 226, 256, 268, 280, 314, 388
- Intermodule text, 333
- Internal data interfaces, 24
- Internal specification, 58
- Interpreters, 44
- Interrupt, 226, 239, 334
- Interrupt-driven modules, 55
- Interrupt handling, 98
- I/O (see Input/output)

- JOIN, 239, 340

- Keywords, 331

- Languages, 174, 187
- Latest finish, 419
- Latest start, 419
- LEAVE, 345, 351
- Let, 344
- Level of abstraction, 40
- Levels of access, 39, 45

546 Index

- Levels of detail, 46
- Lien, 227
- Life cycle costs, 34
- Literal values, 93
- Logo, 360
- Look-ahead, 33, 39, 45, 80
- Loop-collecting nodes, 327
- Loop structure, 345
- Lust, 513
- Macro, 87, 91, 349
- Magie numbers, 93
- Maintenance, 154, 407
- Management, 3, 20, 58, 101, 232, 513
 - data, 212
 - information, 251
 - visibility, 34
- Master copies, 83, 113
- MBASICTM, 25, 26, 86, 95
- Memory usage, 93
- Methodology, 172
- Milestones, 8, 111
- Mnemonic, 31, 63, 76, 82
- Mode, 24
- Module, 60
 - cohesion, 13, 39
 - coupling, 39
 - identifier, 227
- Monitor, progress, 11
- Mobility, 177
- Multi-stage testing, 128
- Named constants, 93
- Naming and referencing, 27
- Narrative, 16, 60, 69, 71, 86, 151, 202
- Negotiations of requirements, 7
- Normal, 350
- Normal terminations, 33, 53
- One-sigma event, 11
- Operational environment, 128
- Operations manual, 383
- Operator manual, 106
- Opossum, 228
- Optimization, 87
- OUTCOME, 339, 347, 350, 351, 354
- Paranormal exits, 53, 92
- Pascal, 183, 195
- Passive voice, 76
- Path monitors, 106
- Peer corroboration, 116
- Perception forms, 213
- Performance measurement, 212
- PERT, 215, 255, 417
- Phase, project, 48, 229, 258
- Planning, 513
- Portability, 177, 189
- Post-order traverse, 208, 229
- Pre-order traverse, 27, 91, 229
- Procedure, 351
 - description language, 311
- Productivity, 184
- Program, 352
 - data base, 177
 - design language, 201, 311
 - design standards, 35
 - labels, 93
 - modes, 16
 - modules, 332
 - real-time, 53, 93, 107, 135
 - segmentation, 47
 - specification, 58, 134, 230, 232, 275, 286
 - standards, 279, 410
 - structure, 36
 - utility, 18
- Programming language, 39, 63, 174, 187
- Progress monitor, 11
- Project archives, 117
- Project control, 83
- Project management, 47, 154, 165
- Project notebook, 36, 77, 112, 113, 118, 141, 154, 166, 373
- Project status, 83
- PS (Programming Specification), 230, 232, 275
- QA (Quality assurance), 114, 115, 154, 165, 513
- Quality, 57
- Quality assurance, 114, 115, 154, 165, 513
- Random number, 499
- Rate chart, 10, 279
- Real-time programs, 53, 93, 107, 135

- Recursive subroutine, 49
- Refined costs and schedules, 13
- Reliability, 186
- Repair effort, 124
- REPEAT, 345
- REQUIRE, 353
- Requirements, 14, 82, 251
 - negotiation, 7
 - standards, 1
- Resource access, 39, 77
- Resource access hierarchies, 49
- Resource protection, 55
- Resource, shared, 55
- RETURN, 342, 351, 353
- Review, 3, 8
- Sample programs, 415
- Scalar types, 195
- Schedule, 6, 8, 216, 376
- Scope of activity, 63
- Scope of variables, 93
- Scoping, 114
- SDL (Software Development Library), 33, 77, 83, 98, 107, 112, 117, 143, 154, 165, 168, 178, 232
- Search, 192
- Secretariat, 174, 178
- Security and privacy, 143, 392
- Semantic, 310
- Sequential testing procedure (STP), 33
- SFS (Software Functional Specification), 14, 58, 134, 232, 275, 283, 309
- Shared resource, 55
- Shared subspecifications, 18
- Side-effect, 231
- Signatures, 7, 33
- SIMULA, 196
- Simulation, 107, 114, 131, 184
- SJR (Software Justification Report)
 - 3, 251
- Slack time, 419
- Software, 232
- Software acquisition plan, 3
- Software Definition Document (SDD), 263
- Software Development Library (SDL), 33, 77, 83, 98, 107, 112, 117, 143, 154, 165, 168, 179, 232
- Software development plan, 46
- Software functional requirement, 3, 46
- Software Functional Specification (SFS), 14, 58, 134, 232, 283, 309
- Software justification report, 3, 251
- Software Requirements Document (SRD), 2, 5, 154, 232, 251
- Software Specification Document (SSD), 7, 9, 15, 17, 45, 46, 58, 77, 85, 94, 111, 134, 154, 232, 263, 275, 309, 416
- Software Test Report (STR), 111, 113, 143, 399
- Software tools, 411
- SOM (Software Operation Manual), 383
- Speed, 93, 158
- SRD (Software Requirement Document), 2, 5, 154, 232, 251
- SSD (Software Specification Document), 7, 9, 15, 17, 45, 46, 58, 77, 85, 94, 111, 134, 154, 232, 263, 275, 309, 416
- Stack, 44, 50
- Standard forms, 141
- Standards waivers, 83, 381
- Statement-continuation, 325, 330
- Status report, 352
 - monitors, 11
- Step numbers, 91
- Stepwise refinements, 40
- STOP, 353
- STR (Software Test Report), 143, 232, 399
- Striped module, 61, 66, 79, 85, 104, 134, 233
 - interfaces, 47, 61
- Striping conventions, 30, 67, 248
- Structural design, 36
- Structure flag, 50
- Structured programming, 45, 313
- Structure types, 195
- Stubs, 94, 100, 105, 109, 191, 354
- Suave, 233
- Subprogram, 66, 79, 335, 351
- Subroutine, 79, 353, 355
 - function, 67

548 Index

- recursive, 49
- Sustaining, 234
- Surviving documents, 20
- Synchronization, 55, 114
- Syntax, 310
- Table-driven algorithms, 44
- Team interfaces, 13
- Team performance, productivity, 6, 10
- Template, 349
- Testability, 19
- Test, 118, 234
 - archives, 137
 - acceptance, 104, 124, 129, 137
 - correctness, 103, 114, 222
 - criteria, 136, 403
 - driver, 118, 234
 - multi-stage, 128
 - philosophy, 107
 - plan, 129, 402
 - policy, 104
 - specification, 129
 - stubs, 105
 - validation, 103, 131
 - verification, 54
- Thrashing, 107
- Tier chart, 40, 46, 76, 82, 83, 203, 216, 320, 325, 365
- Tier number, 46
- TO, 351
- Tolerances, 13
- Tools, 171
- Top-down readability, 50
- Topological sorting, 27, 235, 419
- Training, 257
- Trap, 235
- Type attributes, 63, 195
- Understanding, 76
- Unstriped modules, 61, 67, 85
- User, 154, 184, 213, 235
- User forms, 213
- User manual, 154, 295
- Validation, 103, 131
- Variances, 13
- Verification, 54, 113
- Vertically striped modules, 27, 63
- Visibility, 61, 373
- WBS (see Work Breakdown Structure)
- WHEN, 94, 350, 355
- Work Assignments, 13
- Work Breakdown Structure (WBS), 8, 10, 12, 46, 216, 236, 267, 377, 417
- Work level profile, 127
- Zero defects, 124